

2nd
Edition

Implementing Design Patterns — in — C# 11 and .NET 7

Learn how to design and develop robust and scalable applications using design patterns



Alexandre F. Malavasi Cardoso



Implementing Design Patterns in C# 11 and .NET 7 2nd Edition

*Learn how to design and
develop robust and
scalable applications using
design patterns*

Alexandre F. Malavasi Cardoso



www.bpbonline.com

Copyright © 2024 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2021

Second published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55517-333

www.bpbonline.com

Dedicated to

My beloved wife:

Paula

and

My daughter ***Myla***

About the Author

Alexandre F. Malavasi Cardoso boasts over 16 years in software development, taking pivotal roles as a technical leader and engineer. He has spearheaded major projects using Microsoft Technologies for prominent companies across South America, Europe, and the U.S. Presently, he serves as the Head of Engineering at Propylon and as a Technical Advisor for Marelo. Alexandre's academic accolades include a postgraduate degree in Business and Systems Analysis and two master's degrees emphasizing Software Engineering with Agile Methods. He has earned multiple Microsoft certifications in Azure and Web Development. An active contributor to the global tech community, Alexandre often speaks at international IT conferences, authors technical articles, and has thrice been recognized by Microsoft as a Most Valuable Professional (MVP).

About the Reviewer

Based in Rabat, Morocco, **Nabil Tadili** stands out as a renowned Lead Software Architect with deep expertise in Full Stack development, predominantly in the realms of .NET and Angular. With an unwavering dedication to pioneering innovations and adhering to industry best practices, he's delved profoundly into these platforms, ensuring he's attuned to their evolving nuances. In addition, Nabil showcases mastery in Node, Svelte, NextJs, Python, and a plethora of databases and demonstrates an exceptional command over CI/CD methodologies, representing a comprehensive embodiment of contemporary software engineering.

At Orange Business, Nabil's key role in driving the shift to a Microservices architecture highlights his forward-thinking approach to technological progression. While actively delving into Machine Learning at the Georgia Institute of Technology, he further accentuates his multifaceted expertise by refining business workflows and leading groundbreaking digital projects. His freelance journey, punctuated by over 150 accomplished projects and a stellar 98%+ client satisfaction rate, underscores his flexibility and unwavering commitment.

Ever driven by the motto to challenge one's limits, Nabil is an ardent advocate for teamwork, innovation, and continuous learning. His passion extends to artificial intelligence, quantum computing, and science.

Acknowledgement

My heartfelt appreciation goes to my family and friends for their steadfast support during this book's creation, with a special nod to my wife Paula and daughter Myla. I extend my gratitude to BPB Publications, whose expertise was pivotal in finalizing this work—a journey enriched by the collaboration of reviewers, technical mavens, and editors.

I must also recognize the invaluable insights of my colleagues and coworkers from years in the tech sector. Their teachings and feedback have been indispensable. Lastly, to every reader who resonated with my book: your enthusiasm and encouragement have truly anchored this endeavor.

Preface

In the dynamic world of software development, the mastery of Design Patterns and the intricacies of the object-oriented programming paradigm can set one apart. This book aims to be your guide through these complexities, specifically tailored for the modern version of C# language and the .NET platform. Our journey spans from the foundational SOLID principles, traces the rich history of the .NET platform, and ventures into the core of Design Patterns, all illustrated through tangible, real-world examples and an intuitive, step-by-step methodology.

Our opening chapters lay the groundwork, elucidating the fundamental concepts of C# and .NET. As we traverse through SOLID principles and the essence of object-oriented programming, we set the stage for the immersive experiences that follow. The book's heart revolves around hands-on examples that emphasize best software development practices, all while setting the stage for the profound Design Patterns which dominate the contemporary market.

Upon turning the last page, you, the developer, will find yourself equipped with a holistic understanding of C# and the .NET platform. More than just knowledge, you'll possess the practical wisdom to implement best practices in real-world scenarios and a versatile toolbox of Design Patterns to tackle the myriad challenges of software development. Welcome to a transformative learning experience.

Chapter 1: C# Fundamentals – Diving deep into the C# landscape, this chapter covers the essential components of the language—from its syntax to control structures. Readers will become well-versed with data types, operators, and the nuances of C# development, setting a solid foundation for advanced topics.

Chapter 2: .NET Fundamentals – This section unveils the

architecture of the .NET platform, detailing its core components, runtime environment, and the Common Language Runtime (CLR). Readers will learn about .NET's diverse class libraries, offering a comprehensive view of this powerful platform.

Chapter 3: Basic Concepts of Object-Oriented Programming in C# – Emphasizing the pillars of OOP—encapsulation, inheritance, polymorphism, and abstraction—this chapter offers a C#-centric view. Readers will grasp the power of object-oriented design, learning about classes, objects, interfaces, and more.

Chapter 4: SOLID Principles in C# – This chapter demystifies the SOLID principles, laying the foundation for designing robust and maintainable C# applications. Readers will understand the importance of each principle, from Single Responsibility to Dependency Inversion, and how they guide software craftsmanship.

Chapter 5: Introduction to Design Patterns – A comprehensive introduction awaits as readers embark on a journey to understand the rationale, significance, and application of design patterns, preparing them to tackle software design challenges with confidence.

Chapter 6: Singleton Pattern in .NET Applications – Exploring the Singleton pattern, this chapter showcases its role in ensuring a class has a single instance, guiding readers through its implementation nuances and real-world applications within the .NET environment.

Chapter 7: Abstract Factory Pattern with Blazor – Diving into the Abstract Factory Pattern, readers will comprehend its importance in abstracting the creation of related families of objects. The chapter highlights its synergy with the Blazor framework, offering practical insights.

Chapter 8: Prototype Pattern with ASP.NET Razor – This chapter delves into the Prototype Pattern, emphasizing its utility in creating object copies within the context of ASP.NET Razor. Through examples, readers will learn about object cloning and its benefits.

Chapter 9: Factory Method Pattern Using New Features on C# 11 – With a focus on the Factory Method Pattern, readers will explore its applicability in object creation. Leveraging C# 11's novel

features, the chapter provides an enriched perspective on this creational pattern.

Chapter 10: Adapter Pattern with Entity Framework Core – This chapter illuminates the Adapter Pattern, illustrating its pivotal role in harmonizing incompatible interfaces. Using Entity Framework Core as a backdrop, readers will discover the pattern's application in real-world scenarios.

Chapter 11: Composite Pattern with ASP.NET MVC – Detailing the Composite Pattern, this section sheds light on managing hierarchies of objects. Within the ASP.NET MVC framework context, readers will grasp its essence and benefits, especially in UI rendering.

Chapter 12: Proxy Pattern with GRPC – The Proxy Pattern takes the spotlight here. Readers will learn its significance in controlling access to objects. The integration with the GRPC framework offers a fresh perspective on managing object communications.

Chapter 13: Command Pattern Using MediatR – This chapter dissects the Command Pattern, emphasizing its utility in encapsulating request-response mechanisms. With MediatR, readers will experience its potency in decoupling classes and managing operations.

Chapter 14: Strategy Pattern Using Azure C# and Azure Functions – Exploring the Strategy Pattern, this chapter showcases its role in defining interchangeable algorithm families. With C# and Azure Functions as the foundation, readers will learn to dynamically select algorithmic implementations in cloud-based solutions.

Chapter 15: Observer Pattern – Concluding with the Observer Pattern, readers will delve into the intricacies of maintaining consistency between objects. The chapter highlights the importance of real-time data synchronization and how the pattern ensures efficient object interactivity and communication.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/b4af92>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Implementing-Design-Patterns-in-C-Sharp-11-and-.NET-7-2nd-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in

touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. C# Fundamentals

Introduction

Structure

Objectives

Tools and Environment Setup

Installing Visual Studio 2022

Installing Visual Studio Code

Introduction to Visual Studio 2022

Introduction to Visual Studio Code

History of .NET

.NET Core Versions

.NET 5, .NET 6 and .NET 7

Introduction to C# Language

Loops, Operators, and Iterators

While statement

Do-while statement

For loop

Foreach statement

Operators

Assignment operator

Error Handling

Namespaces

New features in C# 10 and 11

Global using directives

Generic Attributes

List patterns

Raw string Literals

New Debugger Features in Visual Studio

Temporary Breakpoints

Breakpoint hover glyph

Conclusion

Points to remember

Multiple Choice Questions

Answers

Questions

Key terms

2. .NET Fundamentals

Introduction

Structure

Objectives

Multi-platform concepts

.NET for cloud development

Azure Storage Accounts

Azure Functions

New features in .NET 6

Hot reload

.NET MAUI

HTTP/3

Custom platform checks

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

3. Basic Concepts of Object-Oriented Programming in C#

Introduction

Structure
Objectives
Classes and access modifiers
Encapsulation
Inheritance
Reusability
Polymorphism
Partial class
Constructor
Static classes
Structs
Interfaces
Conclusion
Points to Remember
Multiple choice questions
Answer
Questions
Key terms

4. SOLID Principles in C#

Introduction
Structure
Objectives
The Single Responsibility Principle
The open-closed principle
The Liskov substitution principle
The interface segregation principle
The dependency inversion principle
Conclusion
Points to remember
Multiple choice questions

Answers

Questions

Key terms

5. Introduction to Design Patterns

Introduction

Structure

Objectives

Importance of design patterns

History of Design Patterns

Design Pattern categories

Creational patterns

Behavioral patterns

Structural patterns

Patterns for Distributed Systems

Microservices design patterns

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

6. Singleton Pattern in .NET Applications

Introduction

Structure

Objectives

Single pattern definition

Single pattern definition

Multiple instances

Modifications for the single pattern

Thread-safe implementation

Dependency injection

Conclusion

Points to remember

Multiple choice questions

Answer

Questions

Key terms

7. Abstract Factory Pattern with Blazor

Introduction

Structure

Objectives

Abstract Factory definition

Abstract Factory scenario

Abstract Factory implementation

Conclusion

Points to remember

Multiple choice questions

Answer

Questions

Key terms

8. Prototype Pattern with ASP.NET Razor

Introduction

Structure

Objectives

Prototype pattern definition

Prototype pattern implementation

Conclusion

Points to remember

Multiple choice questions

Answer

Questions

Key terms

9. Factory Method Pattern Using New Features on C# 11

Introduction

Structure

Objectives

Factory Method definition

Factory Method implementation

New features in C# 11

Required fields

UTF-8 string literals

Newlines in string interpolations

List patterns

Raw string literals

Conclusion

Points to remember

Multiple choice questions

Answer

Questions

Key terms

10. Adapter Pattern with Entity Framework Core

Introduction

Structure

Objectives

Adapter pattern definition

Adapter pattern implementation

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

11. Composite Pattern with ASP.NET MVC

Introduction

Structure

Objectives

Composite pattern definition

Composite pattern implementation

Conclusion

Points to remember

Multiple choice questions

Answer

Questions

Key terms

12. Proxy Pattern with GRPC

Introduction

Structure

Objectives

Proxy pattern definition

Proxy pattern types

Proxy pattern structure

Common use in real projects

Proxy pattern implementation

Scenario requirements

Practical example

GRPC with Blazor

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

13. Command Pattern Using MediatR

Introduction

Structure

Objectives

Command pattern definition

Proxy pattern implementation

Extra classes

MediatR and Command Pattern

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

14. Strategy Pattern Using Azure C# and Azure Functions

Introduction

Structure

Objectives

Strategy pattern definition

Estrategy pattern structure

Strategy pattern implementation

Creating the city strategy class

Strategy pattern and Azure Functions

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

15. Observer Pattern

Introduction

Structure

Objectives

Observer pattern definition

Observer pattern implementation

Creating the User class

Creating the Blog Post class

Client application

Conclusion

Points to remember

Multiple choice questions

Answers

Questions

Key terms

Index

CHAPTER 1

C# Fundamentals

Introduction

With this chapter, we are starting our journey into design patterns using C# 11.0 and .NET 7, walking through the basic concepts of programming in C#, giving you familiarity with the language, its main operations, and instructions that will help you understand what you need to learn to make progress in the next sections of this book, such as object-oriented programming, design patterns, and .NET platform.

You will learn in this chapter how to create and work with variables, operators, and logical and conditional statements. Additionally, you will have the opportunity to have practical experience in implementing basic programs in C#. Further, you will be able to try the new Debugging Experience provided by Visual Studio 2022.

Getting familiar with the basic concepts of C# will allow you to understand how to apply the complex design patterns in the further chapter in this book. It will help you to build enterprise projects using the .NET platform.

Structure

In this chapter, we will discuss the following topics:

- Visual Studio 2022 and Visual Studio Code Installation Instructions
- Introduction to Visual Studio 2022 and Visual Studio Code
- Basic operations in C#
- Object Types in C#
- Loops and iterations in C#
- Error handling in C#
- New features introduced in C# 10 and C# 11

Objectives

In this unit, you'll learn to set up Visual Studio IDE and Visual Studio Code, create C# apps, grasp basic language operations, and employ new features from C# 10 and 11. These skills will provide a strong foundation for proficient C# programming and development.

Tools and environment setup

To get started with software development in .NET 7 and C# 11, you must install the latest Visual Studio 2022 version, a complete **Integrated Development Environment (IDE)** for creating, compiling, and building your .NET projects. Visual Studio is available for Windows and macOS, both available in the Community Edition for studying purposes. The tool can be downloaded from the official Visual Studio website.

Furthermore, Microsoft has provided the Visual Studio Code, a free, open-source alternative light version of code editor for .NET and C# applications, which is available not only for Windows and macOS but also for various Linux distributions. Considering this editor is an open-source, extensible project, technical communities, IT professionals, and companies around the globe have created numerous extensions for different languages apart from C# itself. Therefore, it is a suitable tool for creating cross-platform applications without any compatibility concerns. Visual Studio Code can be downloaded on the official website for free.

Installing Visual Studio 2022

After downloading Visual Studio 2022 from the official website, you must take the following steps for the installation:

1. Make sure your user has the necessary permissions to install the software on the operating system. Double-click the downloaded executable file.
2. Choose the desired workloads to be installed and set up with Visual Studio, including the additional native project templates from the .NET platform. For the examples of this book, the following workloads must be installed, as seen in [Figure 1.1](#):

- ASP.NET and web development
- Azure Development
- Node.js development
- Mobile development with .NET
- .NET desktop development
- Universal Windows Platform development

After making this step in the installation, the workload list will be shown as the following result in [Figure 1.1](#):

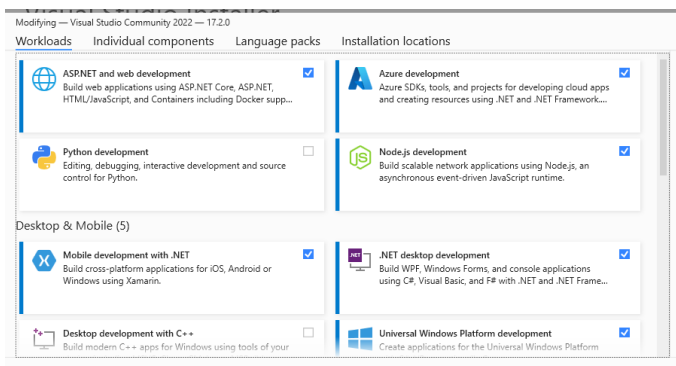


Figure 1.1: Visual Studio 2022 workloads

9. After choosing the necessary workloads, click on the Install option.

10. Usually, Visual Studio is configured to use the same language and region settings as the operating system. If you want to setup a different one, you can do that in the **Language Packs** option, where other ones will be available. Visual Studio was configured to the English language for the examples in this book by default. After finishing the installation, you will be already able to create .NET 7 solutions using all the project type available in Visual Studio that belongs to the workloads specified during installation.

Once Visual Studio 2022 is installed, you do not need to install the latest stable version of the .NET SDK, as it is already a part of the Visual Studio installation. By default, Visual Studio updates and follows the newest features introduced into .NET, such as library updates, minor and major changes, and new project templates. However, any new library updates after the initial installation will not automatically apply the upgrades to your existing projects. Each project targets a specific .NET version, and Visual Studio updates and modifies just the IDE, not the existing projects' configurations.

Installing Visual Studio Code

Visual Studio Code is a cross-platform alternative to Visual Studio IDE, and it is a good option if you want a free, lightweight cross-platform editor for .NET projects. Many companies, technical communities, and individual contributors have provided extensions that allow us to work with many different programming languages in Visual Studio Code. This light open-source IDE has become one of the most popular code editors used in the market. Furthermore, the editor is available for multiple operating systems, such as macOS, Windows, and multiple Linux distributions.

After downloading the executable from the official Visual Studio website, you must take the following steps to complete the installation:

1. Double-click the downloaded executable file. Ensure your user has the necessary permission in the operating system to install the software.
2. Download the Visual Studio extension for C# and Azure, as seen in the following figure:

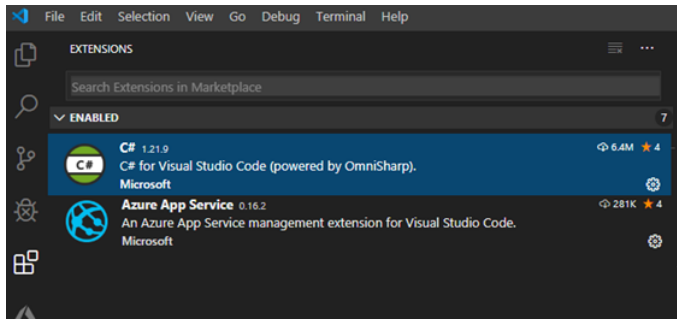


Figure 1.2: Visual Studio Code extensions

These are the two extensions that can be used along with the code samples in this book, and Visual Studio Code can be used as an alternative to the Visual Studio 2022 IDE if you would like to use more command lines for .NET applications.

Introduction to Visual Studio 2022

The Visual Studio 2022 is such a powerful IDE for software development, which allows you to create, build, debug and deploy your .NET applications using a single tool with numerous plugins and extensions, giving you a rich user experience as a developer. It includes the possibility of having access to external resources such as databases and cloud services. In addition, the Visual Studio contains natively installed project templates along with the initial setup, including code samples for Web projects, Desktop, and Mobile applications, which give us a quick way to start a project without having to set up everything from scratch in terms of simple and common configurations. To access those templates, click the **File** option on the super menu and choose the **New Project** option, as seen in [Figure 1.3](#):

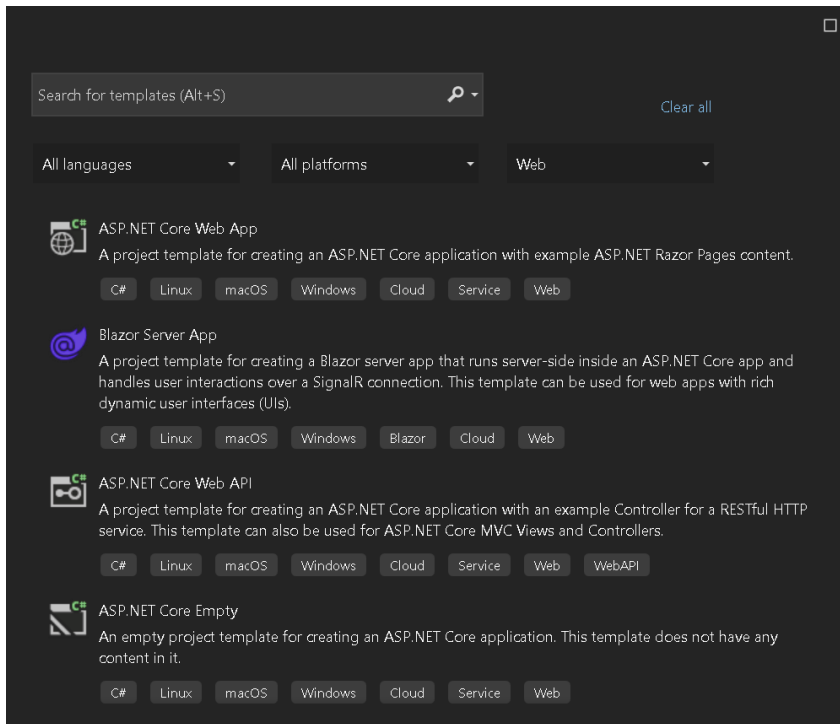


Figure 1.3: Visual Studio templates

After installing the most recent Visual Studio version, the most popular project templates for .NET applications become available, according to the workloads chosen during the installation, such as Console App, ASP.NET Core Web API, Blazor App, and much more. Each project template has other sub-types ready to use, a variation of the main template. Those templates are time-saving, speeding up the software development process and helping learner developers understand how each type of project can be structured.

Open-source communities, companies, and individual developers who publish the templates as Visual Studio extensions on the Visual Studio Marketplace website develop and share many extra free templates. Additionally, Visual Studio extensions for many other purposes can be downloaded on the same website to get a better experience for software development within Visual Studio, getting integrations with third-party tools.

After creating any simple project, the template list available in

Visual Studio, you are redirected to the Solution Explorer experience, having in a common place the code editor, access to the files and folder structure for your project, and access to any relevant external resources, as seen in *Figure 1.4*:

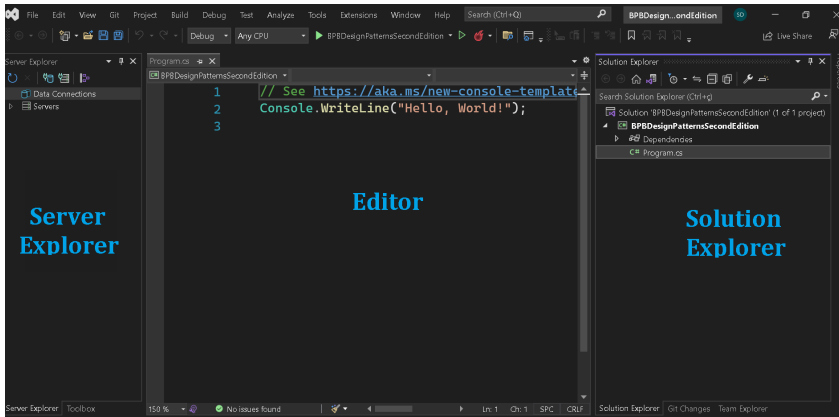


Figure 1.4: Visual Studio Code editor área

On the **Solution Explorer** sidebar, you can access all the existing folders in files of your projects, including all the resources related to all the linked projects if you have multiple projects in the same solution. Furthermore, this **Solution Explorer** view allows you to have access to extra options for the configuration of your projects, such as the installation of extra packages, third-party libraries, and other resources available through NuGet Packages and project references by right-clicking on an individual project in the solution and choosing the **Manage NuGet Packages** option, as seen in *Figure 1.5*:

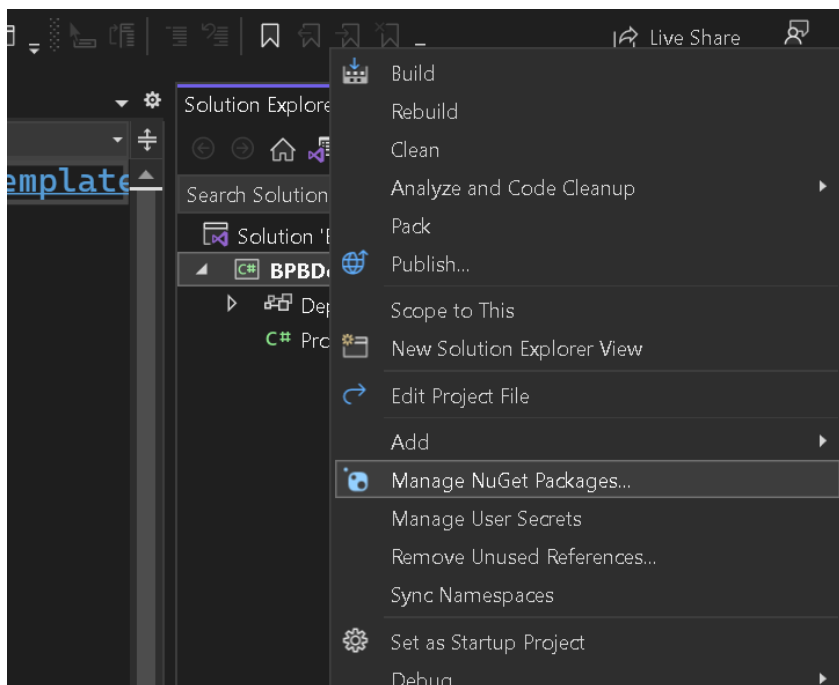


Figure 1.5: Manage NuGet Packages option

After choosing the underlying **Manage NuGet Packages** option, a new window is available where it is possible to search and install any published Nuget packages, specifying the desired version, as seen in [Figure 1.6](#):

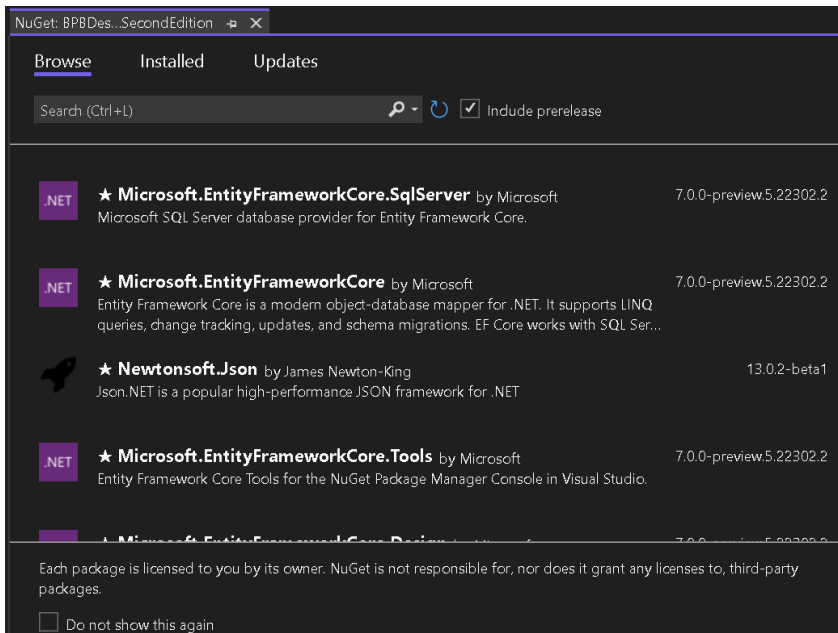


Figure 1.6: Nuget Packages window

Another exciting option is to install packages using the commandline, which makes the process much easier and faster once the developer has enough familiarity with the available commands. Under the Nuget Packages Console window, which appears at the bottom of Visual Studio, it is possible to use the `dotnet install` command, followed by the name of the package, as seen in [Figure 1.7](#):

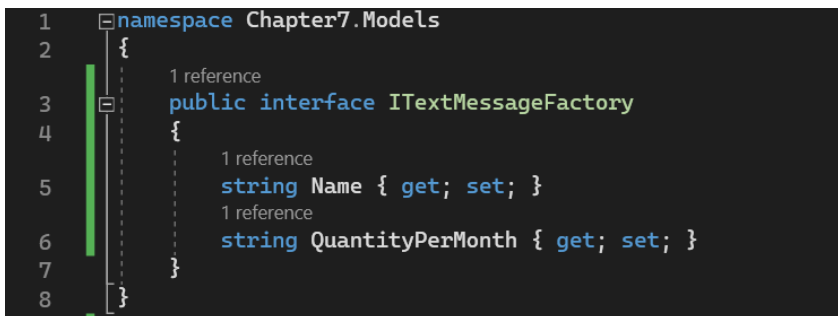


Figure 1.7: Dotnet CLI for Nuget Packages

Continuing the Visual Studio coding area presented in [Figure 1.4](#),

the Editor sidebar is where you develop your code. It can get tips by Visual Studio on code syntax, indentation, codification error messages, navigating into classes, functions and functions and methods, including project references within the source code. The editor has a high level of customization, meaning many settings can be changed to adjust according to your needs. This customization involves background color, contrast level, font size, and more.

Finally, on the **Server Explorer** sidebar shown in [Figure 1.4](#), it is possible to connect to external resources, including databases, application servers, cloud infrastructure, and on-premise features. With all the integrated features in Visual Studio, you can keep all the work in a single and shared place, which has an excellent value for high productivity. In the select Toolbar, there is an option to run applications, save pending changes, hot reload, debug capabilities, comment features, options to undo recent changes, and much more, as seen in [Figure 1.8](#):

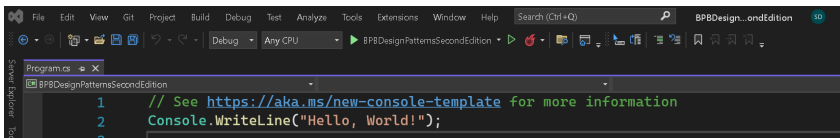


Figure 1.8: Visual Studio Toolbar

The .NET platform allows us to build applications using C#, Vb.Net, and F# languages. F# is a programming language based on the functional programming paradigm, and the Vb.NET is a programming language largely used in .NET projects since the beginning of the platform, despite the fact the C# language has taken over the preference of .NET developers in general to build .NET enterprise applications. Within Visual Studio, under the project creation dialog, you can choose the language for your project by choosing the correct and corresponding project template, as seen in [Figure 1.9](#):

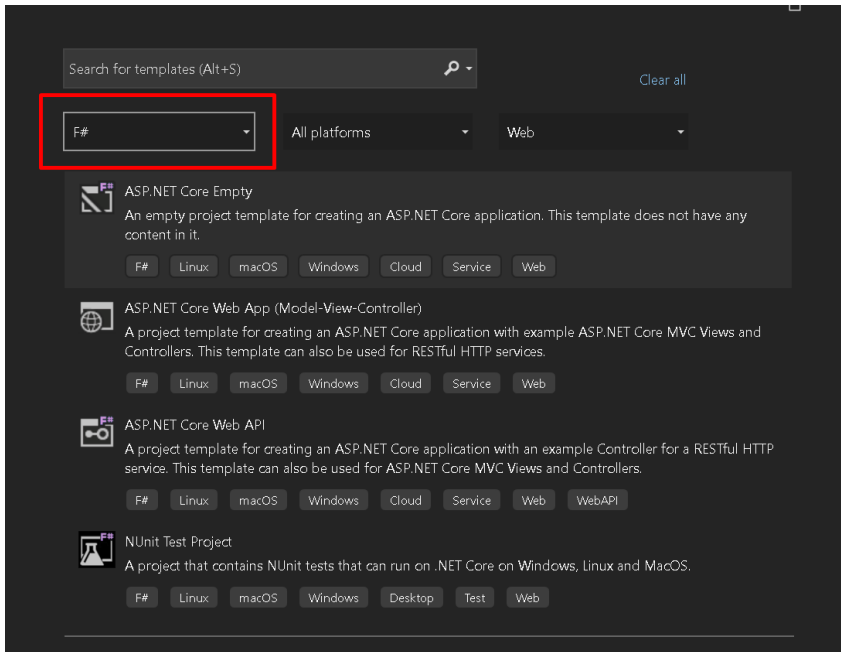


Figure 1.9: Visual Studio templates for F# language

Given the example in [Figure 1.9](#), the Visual Studio shows all the available project templates based on the F# language. Therefore, there are alternatives to the C# language within the .NET platform, including the possibility of multiple projects in the same solution, each targeting a different programming language as the core language.

Visual Studio is a complete IDE for .NET applications. It does not require the installation of extra tools to build and execute standard programs based on C# or any other language supported by .NET.

Introduction to Visual Studio Code

At first glance, Visual Studio Code seems quite different from Visual Studio 2022 in terms of user experience. Still, both have the same purposes: to create, debug, build and deploy applications using the .NET platform or other platforms. At its core, this light IDE contains many extensions to make our routine as developers easier and more productive. It is certain to say that the main difference between Visual Studio Code and Visual Studio 2022 is the cross-platform capabilities for development once the first one is available for

Linux, Windows, and macOS operating systems. Still, Visual Studio is available only for Windows and macOS, with limited capabilities in the macOS version. On the other hand, you must install and enable each extension and extra component on Visual Studio Code to get everything you need to develop specific applications. In terms of comparison with other Visual Studio versions, the Figure 1.10 shows that the options available for Visual Studio Code are pretty similar to the ones present in Visual Studio 2022:

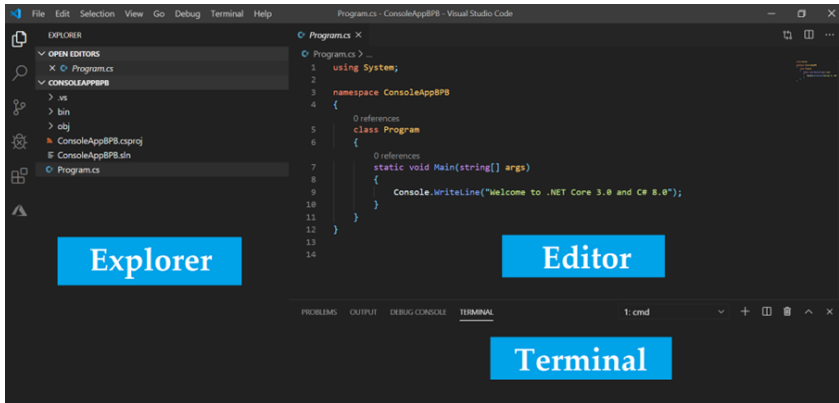


Figure 1.10: Visual Studio Code editor áreas

On the **Explorer** sidebar, you can access all the existing folders and files of your projects, including the possibility of managing multiple and simultaneous projects in various windows in the same Explorer. As seen in [Figure 1.10](#), the **Editor** sidebar is where you develop the actual code and can get a similar experience as Visual Studio regarding code tips, snippets, and much more.

Visual Studio Code is a command-line-based editor, meaning it may have fewer visual options in the IDE to manage and develop your projects. However, you can benefit from quickly running, building, and creating new files for your application using commands without having to use the mouseclicks for it, increasing productivity if you are familiar with the commands and shortcuts available.

The traditional complete Visual Studio 2022 IDE version relies on the existing specification within `.sln` (solution) files and specific extensions for projects, such as `.csproj` files for C# projects, to open the project solutions correctly. Although, Visual Studio Code

allows you to run code by only opening simple folders and files, which is useful in case you want to run other languages that are not .NET related in the same Visual Studio Code simultaneously.

Speaking of productivity and menu options, Visual Studio Code is more flexible and customizable than Visual Studio, as it contains many predefined commands and shortcuts to get files, a more friendly search option across folders and files, and much more, as seen in [Figure 1.11](#):

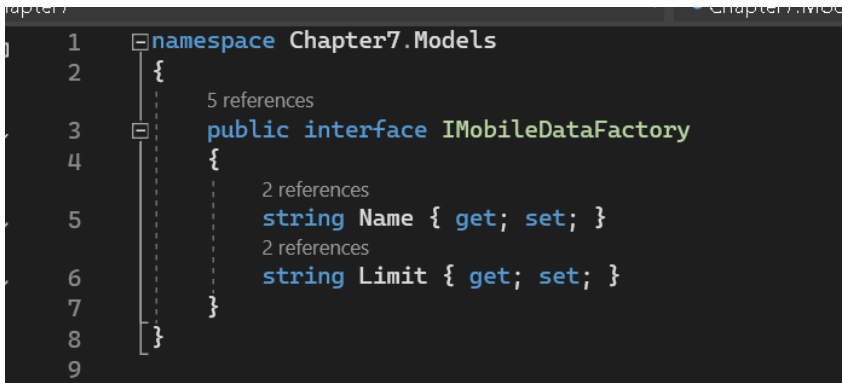


Figure 1.11: Visual Studio Code shortcuts

Visual Studio is the most popular code editor in the market right now. The features available are increasing exponentially over time because of the intense contributions by active technical communities behind this project, providing a lot of extensions and new capabilities.

History of .NET

The .NET platform was created long before Web applications became popular and even before the first C# language. Despite the massive adoption of .NET these days, there is a first impression that the scenario in the market was always like the current state, especially for those who did not experience software development around the beginning of the 1990s. Looking back at almost three decades ago, when other programming languages were prevalent like Java and PHP at this time, it seemed essential for Microsoft to enter this great market of development tools for enterprise applications. In this context, the first version of the .NET platform

was launched as a beta product in 2000. In February of 2022, the official 1.0 version was finally released, supporting Windows 98, Millennium, and XP.

The most important feature of this first version was the **Common Language Runtime (CLR)** capability, which allowed developers to create .NET applications using more than one programming language within the same solution. The .NET platform was able to execute the multiple projects in runtime using CLR, converting the output for each programming language into **Common Intermediate Language (CIL)**, also called **Intermediate Language (IL)**, therefore giving the possibility of sharing libraries written in C# and libraries written in Visual Basic for a single application, increasing the capabilities in terms of reusability. This ability to use multiple programming languages to develop a single application did not necessarily mean that it was possible to create multi-platform applications using .NET at this time. In contrast, running and developing a .NET application only on Windows OS was possible.

In the second quarter of 2003, the first relevant major version of .NET was released under the .NET 1.1 version, and the support for **Open Database Connectivity (ODBC)** was introduced at this stage, following standard requirements for database integrations in .NET applications. Up to this point, .NET did not have wide adoption in the market, and important features should be released to meet the most modern software development demands and trends of this specific time.

In January 2006, Microsoft launched the .NET Framework 2.0 and other relevant tools and products, such as Visual Studio 2005, a more stable release of the new SQL Server 2005, and Biz Talk. One of the most significant achievements of this version was the support of 64-bits computer architecture and new features in C# language, such as partial class, new authentication options for ASP.NET, and Datable objects, which allowed the management of database operations using in-memory databases.

Alongside these new features, Visual Studio was becoming mature and complete for enterprise applications development, being a full IDE, even for programmers who had expertise with other programming languages that .NET did not have support at the time. Additionally, the ASP.NET Web Forms project started having an

extensive adoption among developers who had experience with Desktop development, having a large migration of professionals to Web development. Visual Studio 2005 already contained features to facilitate the creation of visual Web components, allowing dragging and dropping visual Web components on the screen, to build Web pages with similar experience developers had with Desktop development.

ASP.NET Web Forms occupied an essential space in the .NET Framework before the existence of the ASP.NET **Model View Controller (MVC)** project was introduced to the platform. Web page navigation based on forms was common at the beginning of the Web. It essentially considered a user sending requests from the Web browser, and the server returns a response, always keeping the state of the entire page, including all the Web controls, on the server. In this model, the server was responsible for processing the HTML and related content, including dynamic data. This approach used the View State model to store the values of all server-side components between each request. Additionally, ASP.NET Web Forms provided the option to use separated files between HTML and the logical code for each page, which was considered an innovation at this time.

In November of 2006, .NET Framework 3.0 was released, including new features related to Desktop development and Web Services. The **JavaScript Object Notation (JSON)** pattern was not used for integrating services at this time. The **Extensible Markup Language (XML)** was the most used alternative for standard communication between systems. The **Windows Communication Foundation (WCF)** from the .NET platform became the official Web Service of .NET. The WCF became obsolete overtime once .NET replaced it with the Web **Application Programming Interface (API)** model, following the most modern pattern adopted in the market after 2006.

One of the benefits of WCF was its integration with native Visual Studio tools, which allowed developers to import third-party endpoints from other Web Services into Visual Studio by informing the Web Service URL, generating the underlying C# classes for the corresponding objects specified in external services, even the services were written using other programming languages. This

helper was focused on interpreting the XML specification on the third-party Web Service to create the corresponding objects and their properties without being necessary to write any line of code. Additionally, the Service Import tool automatically generated the methods in C# language to establish a connection with the referenced Web Service, allowing to send requests to the service following the underlying service specification and parsing the service response to a format the C# classes would understand. It represented a productive tool at the time and was a key point for WCF to become so popular in the market combined with other .NET tools.

These days, many companies still have Web Services developed using the WCF specification for legacy projects. The same situation happens for ASP.NET Web Forms applications, as many companies adopted it to large and complex projects and have not migrated them yet to a new version of Web projects based on .NET.

Another important technology introduced to .NET Framework 3.0 was the **Windows Presentation Foundation (WPF)** project type, a new way to create Desktop applications using the **Extensible Application Markup Language (XAML)** to develop rich user interfaces using 3D computer graphics capabilities. This type of project represented an alternative to the traditional Windows Form project type existing in the .NET platform.

When the .NET Framework 3.5 was released in November of 2007, it brought performance improvements for Desktop applications. It introduced the first version of Entity Framework, giving the possibility of improving the experience for developers to communicate applications with relational databases, representing an alternative to the **Object-Relational Mapping (ORM)** frameworks that became popular on other platforms. However, a new release of Entity Framework had to be launched in 2008 at the same time as the .NET Framework Service Pack 1, including relevant updates after technical community feedback, with this version not being mature enough to be mainly by a relevant number of companies and projects worldwide. The ORM concept and its importance were widely accepted on the market, and huge improvements to Entity Framework became extremely necessary, mainly to give reliability and stability to the tool.

After only one year of the Entity Framework updates, Microsoft provided version 4.0 of the .NET Framework. Still, it did not present any break changes to previous versions and did not have many new features as part of the update. Differently, the .NET Framework 4.5, released in August of 2012, contained many improvements for Web applications, Desktop development, and native components for Windows 8. Among all the new features in this version, the new capabilities for the ASP.NET application stood out, including support for HTML 5 for Web Forms and support for Websocket protocol, representing a large step to follow the most modern Web standard practices at that point. Additionally, this .NET version allowed the use of asynchronous HTTP requests and responses.

In July 2015, the .NET Framework 4.6 was launched, being the last big version of the framework before it became multi-platform with .NET Core, although it still has minor updates in the versions 4.7 and 4.8. The 4.6 version introduced the support for Azure SQL Server distributed transactions, improvements related to the WPF user interface, and support for data localization using annotation attributes for ASP.NET applications.

.NET Core versions

Considering the existing scenario back in 2015, the market for software development was changing to become compatible with cross-platform concepts. In this context, the fact that the .NET platform had support only for Windows operating systems represented a significant limitation for its evolution and influence in the market. In 2005, it was largely accepted by companies, developers, and technical communities that cross-platform development was something mandatory for the success and costs of any software project, with the capability for the underlying technologies involved in the software development process to be executed in multiple operating systems, multiple types of devices, and to have interoperability with other technologies and platforms.

Focused on meeting these new requirements, Microsoft launched in June of 2016 the first version of .NET Core. This disruptive platform allows the development, deployment, and execution of .NET applications on Linux and macOS. The .NET libraries and packages for C# and Visual Basic languages were completely re-written from

scratch, and a new version of ASP.NET projects and Entity Framework were created as well, with these products being renamed to ASP.NET Core and Entity Framework Core, following the new brand of .NET for cross-platform development.

Additionally, all the .NET Core libraries and their projects became open-source projects, with the code being shared on Github, making it possible for anyone in the technical community to contribute and collaborate to the evolution of the subsequent releases, reporting issues on features and giving feedback on new capabilities within the platform on various projects. The change to an open-source model represented a big milestone for .NET developers, as only the ASP.NET MVC project could be considered an open-source project at this point. It significantly contributed to .NET adoption by developers from other technologies.

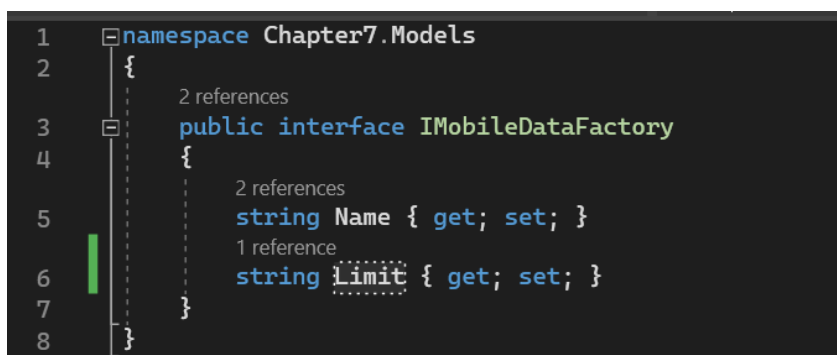
.NET Core 1.1 was released in November of 2016, containing many solutions for issues reported by developers and companies, and focused on supporting a more comprehensive range of Linux distributions. Additionally, this version improved the Entity Framework tool to support Azure and SQL 2016.

In August of 2017, .NET Core 2.0 was launched, with the possibility to integrate with legacy .NET Framework libraries using .NET Standard as middleware, improving cross-platform support with new features. The version brought the new Razor Pages Web project, an alternative to the traditional ASP.NET MVC project type for creating Web applications within the .NET Core platform. At this point, .NET Core became a mature platform, primarily adopted by companies worldwide. The support for cross-platform development was the key of .NET Core, joined with Visual Studio Code and new command-line options. Developers were able to create .NET applications that could be executed almost everywhere, taking part in the evolution of the next version of .NET with incremental solid updates via open-source projects, also taking the benefits of performance improvements across many .NET libraries.

The .NET Standard mentioned previously in this chapter was a typical API specification for .NET libraries that allows us to use it across all .NET projects, regardless of its version. This means that different types of projects could share libraries with the .NET Standard, including compatibilities between a wide range of .NET

Framework and .NET Core libraries. It is essential to mention that .NET Core and .NET Standard are different things. The first one is not directly compatible with .NET Framework libraries, and the second one was made to be a bridge between all .NET versions. Therefore, the best strategy to migrate .NET Framework applications to .NET Core is to create libraries in .NET Standard. In that way, it is possible to keep the implementation of legacy projects without any significant breaking changes and use the same libraries in .NET Core projects to transition to a new version of .NET gradually.

As you can see in [Figure 1.12](#), .NET Standard represents the intersection between .NET Framework and .NET Core libraries, keeping high-level compatibility in the version 2.0, reaching a good point of maturity:



```
1 namespace Chapter7.Models
2 {
3     2 references
4     public interface IMobileDataFactory
5     {
6         2 references
7         string Name { get; set; }
8         1 reference
9         string Limit { get; set; }
10    }
11 }
```

Figure 1.12: .NET Standard

Each new version of the .NET Standard added universal compatibility for .NET projects, including .NET Core, .NET Framework, Mono, Xamarin, and Unit. .NET Standard represents a base specification for any library across the entire .NET ecosystem, including Mobile, Desktop, Web and Game development. The .NET Standard 2.0 supports interoperability with .NET Framework 4.6.1 or later versions. Therefore, projects using an older version of the .NET Framework would not be able to use .NET Standard libraries target the 2.0 version. In that case, it is recommended to migrate legacy projects to .NET Framework 4.6.1 before starting any migration to .NET Core.

Since the first version, .NET Core had many improvements and

significant new features, with the cross-platform capabilities being strictly followed in all the versions, including the .NET Core 3.0 version, being a game-changer in terms of compliance with the requirements of modern software development as it supports Windows Forms, WPF, **Artificial Intelligence (AI)**, and **Internet of Things (IoT)**. In the following chapters of this book, you will have the opportunity to dive deeply into all these types of projects and understand the performance improvements that were applied to .NET Core, .NET 5, .NET 6, and .NET 7.

.NET 5, .NET 6 and .NET 7

In November 2020, Microsoft released the .NET 5, which represented the beginning of a new era in software development using the .NET platform once all the effort for cross-platform development reached a good level of maturity and stability. This means that the most relevant .NET libraries were fully migrated to have cross-platform compatibility at this point.

Among all .NET versions, .NET 5 represents an evolution of .NET Core 3.1 and .NET Framework 4.8, a massive milestone in consolidating all the effort regarding cross-platform development that started in the .NET platform since .NET Core 1.0. Therefore, it is possible to say that .NET 5 is a unification of the .NET ecosystem for modern and cross-platform development, combining new features and relevant performance improvements.

The unified platform has the cross-platform focus as a key point, which does not only mean compatibility across multiple operating systems, but in terms of multiple devices and platforms, such as Desktop, Web, Cloud, Gaming, IoT, and others. The achievement of multi-platform development using .NET required a complex re-architecture of the entire .NET platform, bringing at the same time significant improvements in terms of performance for existing features for C# language, including type conversions, parallelization, and much more.

In November of 2021, the .NET 6 was launched by Microsoft, which included further performance improvements, Arm64 support, .NET MAUI, C# 10, new project templates, support for HTTP/3, improvements on code analysis, DateTime and timezone improvements, and much more. At this point, many companies were

already using .NET 5, and the migration to .NET 5 did not represent a huge challenge for legacy projects.

Finally, in November of 2022, .NET 7 will bring significant improvements for modern client development with .NET MAUI, Cloud Native improvements, and new tools to upgrade existing .NET applications.

Introduction to C# language

The best way to learn any programming language is to create real applications for practicing and apply the technical knowledge regarding the most basic concepts of coding in real scenarios. First, basic concepts apply to any programming language, such as variables, loops, blocks, and iterators. In the following sections of this chapter, you will have the opportunity to experience and create multiple C# applications, having exposure to most concepts of the language.

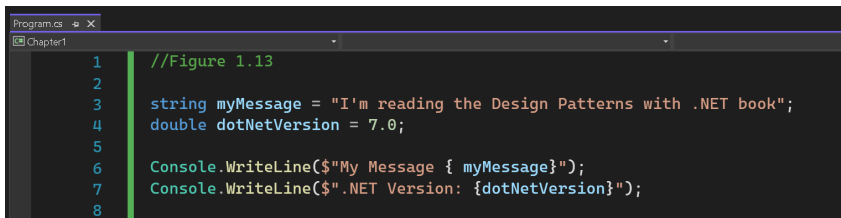
C# is a strong-typed language requiring all the variables specified with a specific type. Therefore, if a variable is created to store only numbers, any attempt to store any other type of content results in an error in compile time. In computer programming, there are also weakly typed languages, which are not pre-compiled, with the variable types being checked dynamically once the application runs. As a strong-typed language, C# language makes it mandatory to specify any variable type in the source code, explicitly indicating what type of data should be stored on each variable.

A variable in a programming language could store simple data structures, such as numbers, strings, and Boolean values. Still, it also could store more complex data that represent custom objects, such as generic list types, structs, arrays, instances of classes, and much more. The correct use of variable types represents an essential aspect of software development in terms of good practices considering they are closely related to the application's state and how it interacts and handles memory and other computer resources. Sometimes, mistakes made on variable specifications may represent a security vulnerability.

In terms of code readability, it is essential to use a good naming convention for variables and any custom implementation during

software development. Always use meaningful names for variables, methods, classes, functions, projects, and structs. In C# language, the camel case pattern for classes is primarily used (for example, **MyBook**) and the upper case for constants (for example, **MAX_ITEMS_ORDER**). Make your code easy to read by others and even by yourself. It increases productivity, reduces project costs, and boosts collaboration in software development teams.

Figure 1.13 represents a simple creation and value assignment for a C# variable within a Console application:

A screenshot of a code editor window titled 'Program.cs' and 'Chapter1'. The code is as follows:

```
1 //Figure 1.13
2
3 string myMessage = "I'm reading the Design Patterns with .NET book";
4 double dotNetVersion = 7.0;
5
6 Console.WriteLine($"My Message { myMessage}");
7 Console.WriteLine($" .NET Version: {dotNetVersion}");
8
```

Figure 1.13: C# variables

Lines 3 and 4 represent two different variables created using different types. The first one uses the string type, which stores the content “I’m reading the Design Patterns with .NET book.”

Considering that it is a string value, the content must be around quotes, as that is the pattern for any string in the C# language. For the second variable in the code sample (line 4), a double type was specified, which stores a decimal value (7.0) and does not require using quotes as it is a number value. The variable should be named according to its purpose, and it can be used in other of the program within the same scope: class, method, or a local function.

There are many variables in the C# language, and each has a specific reason to be used. The following list contains the most popular types and their detailed explanation with examples:

- **object**: It is the most basic type in C# language because all the other types are derived from it. This type allows us to assign values of any type as a generic type, primarily used for dynamic types, mainly in cases where the variable’s target value is unknown.
- **int**: This type was created to store integer numbers, positive or negative. Similar to any other variable type, there are limits

in terms of size established for integers (maximum and minimum values) that can be stored. In that case, the minimum value for integer in C# language is -2,147,483,648, and the maximum value is 2,147,483,647. This limitation helps to prevent memory leaks and security vulnerabilities. Each time a variable is created in the program, a space is reserved in the memory to store the value and keep the reference in the memory for reference types. For this reason, choosing the suitable variable type for any variable used in the application is highly recommended. Furthermore, it is possible to use integer types with other sizes in the C# language. For instance, the short type allows us to store a number between -32,768 and 32,767 (**System.Int16**), the long type allows us to store an integer between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (**System.Int64**), and there are many more options. The correct integer type to use in a specific program depends directly on the requirements of the data in terms of precision and size. Usually, the int type is used in most cases if there is no explicit requirement determining another type, with this type representing a 32-bit number (**System.Int32**).

- **boolean**: This variable type has the purpose of storing logical content that represents a **true** or **false** value, as shown in [Figure 1.14](#):

```
11  
12     bool featureEnabled = true;  
13     Console.WriteLine($"Feature enabled: {featureEnabled}");
```

Figure 1.14: C# bool variables

Line 9 represents a bool variable string, a **true** value, and line 10 prints the value for the underlying value on the console.

- **string**: As shown in [Figure 1.8](#), the **string** type has the purpose of string a sequence of Unicode characters and requires the use of quotes around the value. The correct syntax for the **string** type indicates to the compiler that the content is effectively a string content. Additionally, it is possible to use the **String** class as an alternative to achieve similar results. Both contain similar properties, but considering the **String** type, with a capital letter, is a complex type, it will be handled differently on the memory in a less efficient way in specific scenarios that requires a structure use of computational resources and relevant scale

workload. Each primitive type in C# language has its corresponding class under the System namespace: **System.String**, **System.Int32** and many others.

- **byte**: It is a type used for storing unsigned integer values that start with “0” (minimum value) and “255” (maximum value). It is commonly used to manipulate file bytes in arrays of that type (byte) and low-level programming.
- **char**: This type stores a single Unicode character, unlike the string type that stores a sequence of those values. The utilization of this type is especially useful to allocate less space in the memory if the content is just a simple Unicode character. If the quantity of characters in a variable is predefined and predictably a Unicode character, choose this type of variable instead of string type.
- **float**: This is used to store numbers with precision between 6 and 9 digits, having a maximum size of just 4 bytes.
- **decimal**: This type is used to store numbers with precision between 28 and 29 digits, and the maximum size is 16 bytes.

Each type is suitable for a specific type of data. For example, the decimal type is primarily used to store financial information, as seen in [Figure 1.15](#):

```
17 decimal dollarValue = 3.5m; //Dollar value
18 decimal price = 103.45m; //Product price
19 float floatValue = 1.07f; //Float value
20 double scale = 67.5; //Double value
21
22 Console.WriteLine($"Dollar value: {dollarValue}");
23 Console.WriteLine($"Price: {price}");
24 Console.WriteLine($"Float value: {floatValue}");
25 Console.WriteLine($"Scale: {scale}");
```

Figure 1.15: Type examples

All the variable types shown in this chapter can be considered primitive types in C# language, as they represent the simplest structure in a program to store data. Each type has its limitation in terms of size and allocation in the memory once the variable is created. The .NET garbage collector feature cleans them up from the memory once they are not used to keep only the necessary information on the memory and to have high-performance memory

management.

If an incorrect value type is assigned to a variable, the compiler shows an error, preventing the program from running. This type-check in compile-time represents a huge advantage in favor of the use of strongly typed language like C# because this feature allows us to figure out programming issues earlier than other languages that accept only dynamic types, which values are verified in runtime.

In C# language, there are differences between value type and reference type variables. The primitive types (**int**, **string**, **bool**, and so on) are considered value types once their value is stored in their own memory space. On the other hand, the reference type has a reference placed in the memory for the variable, and its underlying value is placed in different memory spaces. Being familiar with these differences is essential when a scenario requires the development of a high-demand system and requires high performance.

Although **string** and **object** types are considered primitive types, they can also be considered reference types in C# language alongside classes, interfaces, and delegate types. Therefore, there is no mandatory relation between value types and primitive types.

The following topics contain the most popular native reference types and their detailed explanation with examples:

- **Array:** This type has the purpose of storing a collection of objects, allowing to specify multiple values in a single variable, optimizing time in the manipulation of a huge amount of values using multiple distinct variables. To create an array, you must declare the variable type followed by square brackets, and the values can be assigned separated by a comma. As seen in [Figure 1.16](#), you can use any available iteration resource in C# (**while**, **foreach**, and **for**) to loop through each value of the array:

```

29  string[] daysOfWeek = {
30      "Monday",
31      "Tuesday",
32      "Wednesday",
33      "Friday",
34      "Saturday",
35      "Sunday"
36  };

```

Figure 1.16: Array in C#

- **List:** It is a complex object that allows us to manipulate elements as a list and access those values by index. This type contains many extension methods to search elements on the list, sorting, and making various operations. This is one of the most used types in C# because almost all the programs have a common need to manipulate lists of elements that come from databases and other sources. When a list is created, the type of elements should be defined already, and the values can be assigned using the **Add** method, which is part of the **List** class. This type supports a wide variety of operations, the main difference between that type and array variables. Both can get similar results in terms of the manipulation of elements. Still, the **List** type contains more advanced options to insert, remove and search items in the collection without explicitly referring to indices in the list manipulation. These extra features for **List** type help us prevent errors in dynamic data manipulation once arrays always have a determined range of assigned values. [Figure 1.17](#) has an example of a List, showing the way to assign elements to the collection:

```

40  List<string> daysOfWeekList = new List<string>();
41  daysOfWeekList.Add("Monday");
42  daysOfWeekList.Add("Tuesday");
43  daysOfWeekList.Add("Wednesday");
44  daysOfWeekList.Add("Thursday");
45  daysOfWeekList.Add("Friday");
46  daysOfWeekList.Add("Saturday");
47  daysOfWeekList.Add("Sunday");

```

Figure 1.17: List in C#

Lists in C# are under the **System.Collections.Generic** namespace and any type can be combined with the list type as it

accepts a Generic type as an argument. A Generic type can be specified as an element type parameter within brackets, as seen in [Figure 1.18](#):

```
0 references
54  public class MyGenericClass<T>
55  {
    0 references
56      public T? MyData { get; set; }
57  }
58
```

Figure 1.18: Generic type class

In the creation of an instance of the **MyGenericClass**, a specific type must be passed as an argument, and the **MyData** property automatically assumes the same type, as seen in [Figure 1.19](#):

```
56  MyGenericClass<string> myGenericClassOne = new MyGenericClass<string>();
57  myGenericClassOne.MyData = "String Value";
58  Console.WriteLine($"This is a string:{myGenericClassOne.MyData}");
59
60  MyGenericClass<int> myGenericClassTwo = new MyGenericClass<int>();
61  myGenericClassTwo.MyData = 10;
62  Console.WriteLine($"This is an integer:{myGenericClassTwo.MyData}");
63
```

Figure 1.19: Generic type example

In the example in [Figure 1.19](#), a first generic object is created, passing the **string** type as an argument, which means that the underlying **MyData** property assumes **string** as type. The same happens with the second object called **MyGenericClassTwo**, which receives the **integer** type as an argument and automatically reflects this type on the related **MyData** property for this object:

- **Class:** In C# language, it is possible to create our custom types, considering it is an object-oriented language. There are many particular concepts related to classes that will be explained in detail in [Chapter 3, Basic Concepts of Object-Oriented Programming in C#](#) in this book, but only to get started with the basic concepts of classes, you must know that class is one of the reference types in C# language, and its correct use and definition is an essential part of any software using C# language. It is possible to define properties, visibility, constructors, and other inherent concepts to object-oriented programming through classes. For now, it is essential

to understand it as one of the available types in C#. [Figure 1.20](#) contains an example of a class called `Book`, which has six properties: `Id`, `Title`, `Creation`, `Author`, `Summary`, and `ISBN`. You must realize that the types for each property correspond to primitive or complex types, depending on the kind of information that should be stored on them. The `Author` property represents a complex type of `Author` class; therefore, it refers to another entity, as seen in [Figure 1.20](#):

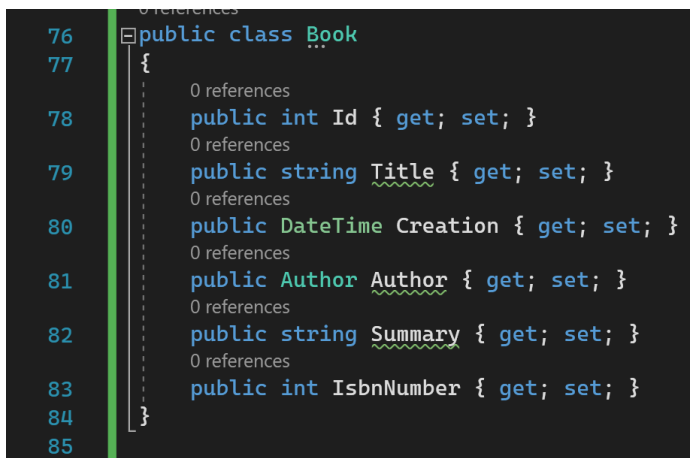
A screenshot of a code editor showing the definition of a C# class named `Book`. The code is displayed on a dark background with light-colored text. Line numbers 76 through 85 are visible on the left. The class definition starts with `public class Book` on line 76, followed by an opening curly brace on line 77. The class contains six public properties, each with a `get` and `set` accessor. On line 78, the `Id` property is of type `int`. On line 79, the `Title` property is of type `string`. On line 80, the `Creation` property is of type `DateTime`. On line 81, the `Author` property is of type `Author`, which is underlined with a red squiggly line. On line 82, the `Summary` property is of type `string`, also underlined with a red squiggly line. On line 83, the `IsbnNumber` property is of type `int`. The class ends with a closing curly brace on line 84. To the left of the code, there is a vertical green line. Above the code, the text "0 references" is visible. To the right of the code, there are several "0 references" labels, one for each property, indicating that no other code in the current context has a reference to these properties or the class itself.

Figure 1.20: *Book class*

- **Other types:** There are other reference types in C#, such as delegates and interfaces, which will be explained in detail in the following chapters. They involved more advanced programming concepts, and many examples can be found in [Chapter 3](#), *Basic Concepts of Object-Oriented Programming*.

Understanding the most basic types existing in C# language allows you to start coding following good practices in terms of memory and other computational resources usage and keep consistency across the application related to naming convention and correct state management for variables and objects.

Loops, Operators, and Iterators

There are many ways in C# language to manipulate lists of objects and execute recursive logical programming to meet specific requirements for the software implementation, reducing significantly the number of necessary lines of code for the routines

required to execute repeated operations in lists or collections. In the following topics, you will have the opportunity to walk through the most common features in the C# language to execute repetitive operations in lists and arrays, understanding the main differences.

While statement

With the **while** statement, it is possible to execute operations till a predefined condition is not satisfied (**false**). The program tests the condition at the beginning of the loop, and if the result is true in the test, the loop will still be executing. Its use is similar to other statements in the C# language for loop operations. Still, the syntax used for the condition, in this case, is clear enough for the developer trying to interpret and debug the code. Therefore, the conditions used to determine if the loop should still run or not are easy to read by the developer as it is at the beginning of the loop. In [Figure 1.21](#), there is an example of a **while** statement, which has a condition to keep running till the condition is not satisfied:

```
65
66     int timesExecuted = 0;
67
68     while(timesExecuted < 25)
69     {
70         Console.WriteLine($"Time(s) executed:{timesExecuted}");
71         timesExecuted++;
72     }
```

Figure 1.21: While in C#

The code in [Figure 1.21](#) has the purpose of printing the message **Time (s) executed** followed by the times the loop was executed. In this case, the loop runs 25 times considering the initial value for the **timesExecuted** variable is 0 (zero), and the same variable is incremented, increasing one number on each iteration, as it is possible to check in line 71 of the same code.

Do-while statement

The **do-while** instruction is very similar to the regular **while** statement, except that the logical code within the block runs before the part of the code that tests the condition. Therefore, the code runs at least once as the logical test is executed at the end of the loop on each iteration. [Figure 1.22](#) contains a code with the same

purpose as the one shown in [Figure 1.21](#) but uses `do-while` instruction instead:

```
77 int timesExecuted = 0;
78
79 do
80 {
81     Console.WriteLine($"Time(s) executed:{timesExecuted}");
82     timesExecuted++;
83 }
84 while (timesExecuted < 25);
85
86
```

Figure 1.22: do-while in C#

This loop is proper when a specific operation must be done before the loop tests the condition, representing the main difference from the `while` statement presented earlier in this chapter. In this simple routine in [Figure 1.22](#), the initial value of the `timesExecuted` variable is **0 (zero)**, and the body of the loop is executed the first time regardless of the condition in the final of the loop, with the iteration happening 25 times. In specific scenarios, the condition tested by the `while` loop could never be satisfied, resulting in an infinite loop by mistake, throwing a **Stackoverflow** exception.

For loop

There are many ways to get similar results in a programming language, and this premise applies to loop operations in the C# language. Apart from the `while` and `do-while` statements, it is possible to get similar iterations in lists using the `for` loop, which allows the execution of repetitive operations when the number of times specified for the execution is satisfied. To use this type of loop, you create a statement following these requirements:

- A variable that contains the initial value for the iterations should be created inside the `for` statement.
- The condition to be tested and satisfied should use the variable created for the initial iteration.
- The times of the executions should be incremental and predefined.

To clearly understand this type of loop, you can see the details in

Figure 1.23, which contains an implementation with similar purposes to the ones represented previously in *Figure 1.21* and *Figure 1.22*:

```
90  for(int timesExecuted = 0; timesExecuted < 10; timesExecuted++)
91  {
92      Console.WriteLine($"Time(s) executed:{timesExecuted}");
93  }
```

Figure 1.23: For loop in C#

As seen in *Figure 1.23*, the `timesExecuted` variable is declared inside the `for` loop statement, and the increment process occurs after the condition that needs to be satisfied, which is in the second instruction of the loop. If you compare it to the previous implementation in *Figure 1.21* and *Figure 1.22*, the `for` loop seems more straightforward and easier to read than the others, despite its same purpose and result. Therefore, there are many ways to reach the same target in logical operations involving lists and loops. It would help if you chose the most suitable option for your enterprise during an implementation.

foreach statement

The `foreach` loop can execute operations in lists of elements without having to access the elements using indices. `Foreach` statements can be used with any type that implements the `IEnumerable<T>` interface in C#. *Figure 1.24* shows how `foreach` works, executing a block of code for each item in a list:

```
98  List<string> daysOfWeekCollection = new List<string>();
99  daysOfWeekCollection.Add("Monday");
100  daysOfWeekCollection.Add("Tuesday");
101  daysOfWeekCollection.Add("Wednesday");
102  daysOfWeekCollection.Add("Thursday");
103  daysOfWeekCollection.Add("Friday");
104  daysOfWeekCollection.Add("Saturday");
105  daysOfWeekCollection.Add("Sunday");
106
107  foreach(string day in daysOfWeekCollection)
108  {
109      Console.WriteLine($"Day of Week: {day}");
110  }
```

Figure 1.24: Foreach in C#

As seen in [Figure 1.24](#), a list of string elements is created in line 98, and the same list is populated between lines 99 and 105. A `foreach` instruction is executed once for each item on the list, specifically between lines 107 and 110. That means the code block that prints **Day of Week** runs seven times, one per day of the week. In general, loop implementations have the advantage of writing less code in case there is a need to run repetitive operations. In this example, instead of specifying the `Console.WriteLine` instruction seven times (one for each day of the week), it was specified just once.

Operators

Operators are primarily used in software programming for mathematical operations, including arithmetic operations, comparison between values, and to specify conditions for Boolean values. C# language contains a huge number of operators. In the following topics, you will have the opportunity to walk through the main types, their usage, and everything, followed by examples.

Arithmetic operators

This type of operator is used to make the addition, subtraction, division, multiplication, and other popular and conventional arithmetic operations. It is possible to specify these operations using variables or directly by the specification of values, as seen in [Figure 1.25](#):

```
115     int number1 = 50;
116     int number2 = 25;
117
118     int totalSubstraction = number1 - number2; //25
119     int totalAddition = number1 + number2; // 75
120     int totalMultiplication = number1 * number2; //1,250
121     int totalDivision = number1 / number2; //2
122
123     Console.WriteLine($"Total subtraction: {totalSubstraction}");
124     Console.WriteLine($"Total addition: {totalAddition}");
125     Console.WriteLine($"Total multiplication: {totalMultiplication}");
126     Console.WriteLine($"Total division: {totalDivision}");
127
```

Figure 1.25: Arithmetic operators in C#

As seen in [Figure 1.25](#), variables are created between lines 118 and 121 to store the total of arithmetic operations involving two numbers defined in the variables `number1` and `number2`. It is

important to note that the variables that store the result of the operations should be created respecting the correct type for the result of the operation. In this context, if at least one of the two numbers had a decimal value, the result variable should be from the decimal type or float type, depending on the precision expected from the arithmetic operation.

Assignment operator

The **equal** operator is the most common one used in C# language, as it is used to assign value to variables, as shown in [Figure 1.26](#):

```
131 string customMessage = "The variable is assigned with this value";
132 int number = 5;
133 int x, y, z;
134
135 x = y = z = 10;
136
137 Console.WriteLine($"Custom message: {customMessage}");
138 Console.WriteLine($"Number value: {number}");
139 Console.WriteLine($"Variable x: {x}");
140 Console.WriteLine($"Variable y: {y}");
141 Console.WriteLine($"Variable z: {z}");
```

Figure 1.26: Assignment operator in C#

In line 131, a string variable called **customMessage** is created, and a value is assigned to it using the operator **=**. The exact process happens in line 132, but with an **int** type. There is the possibility to assign a value to multiple variables in sequence using the equal operator, which means that all the variables will have the same value in the final, which is demonstrated in line 135, where all the three variables (x,y, and z) has the number 10 as value. Therefore, the equal operator in the C# language is not necessarily used only for arithmetic operations but also to set value to variables. Variables are always placed on the left side in this case and the underlying value on the right side.

The **equal** operator is frequently used as well for conditional and comparison statements, but using two times the equal operator instead, as seen in [Figure 1.27](#):

```
146     DateTime today = DateTime.Now;
147
148     if(today.Day == 4)
149     {
150         Console.WriteLine("This is the day number four");
151     }
152
153
```

Figure 1.27: Equal operator

In the example given, line 148 in [Figure 1.27](#) uses the operator `==` to verify if the Day property equals the number 4. Therefore, the equal operator has a different meaning in this context, not as an assignment operator but used for comparison.

Relational operators

One of the most common logical implementations in any programming language involves Boolean expressions, as any system could have routines that need to follow specific logic based on conditions. To achieve this target, the C# language contains many relational operators that allow us to compare two or more values, from the most straightforward operation to the most complex ones, according to what is needed. The following table contains all the relational operators that can be used in the C# language:

	Example	Result	
Equal to	25 == 25	True	
Not equal to	25 != 10	True	
Less than	25 < 45	True	
Greater than	25 > 5	True	
Less than or equal to	25 <= 25	True	
Greater than or equal to	25 >= 25	True	

Table 1.1: Relational operators in C#

Regarding the equal operator, in C# language is essential to note that to make a comparison between two values, two equal operators should be used. In that way, the compiler recognizes it as a comparison and not as a value assignment. Additionally, the `!` operator can be used to reverse the Boolean value for any statement. [Figure 1.28](#) contains examples of each relational operator

in C#:

```
157     DateTime currentDay = DateTime.Now;
158
159     if(currentDay.DayOfWeek == DayOfWeek.Monday)
160     {
161         Console.WriteLine("Today is Monday");
162     }
163
164     if(currentDay.Day != 25)
165     {
166         Console.WriteLine("Day is different from 25");
167     }
168
169     if(currentDay.Day < 25)
170     {
171         Console.WriteLine("Day is less than 25");
172     }
173
174     if (currentDay.Day > 25)
175     {
176         Console.WriteLine("Day is greater than 25");
177     }
178
179     if (currentDay.Day <= 25)
180     {
181         Console.WriteLine("Day is equal or less than 25");
182     }
183
184     if (currentDay.Day >= 25)
185     {
186         Console.WriteLine("Day is equal or greater than 25");
187     }
```

Figure 1.28: Relational operators

Relational operators can have an inverse result if the **!** operator is used before any relational statement or condition. This is especially useful to give more readability to the code when an original relational condition returns **false** by default.

Logical operators

Logical operators are used to combining multiple relational operators in compound conditions. They are just Boolean statements that return **true** or **false** values. This type of operator is generally used in loops and decision statements within logical implementations when one or more conditions in a sentence should be satisfied.

The **AND** and **OR** operators can be combined to make different comparisons, and depending on their order, the statement can return different results. For example, imagine a program with a specific operation that should be executed only if the day of the week is Monday or Tuesday. The **OR** operator can be used in that case to combine two logical expressions, as seen in [Figure 1.29](#):

```
193     var currentDayOfWeek = DateTime.Now.DayOfWeek;  
194  
195     if(currentDayOfWeek == DayOfWeek.Monday || currentDayOfWeek == DayOfWeek.Tuesday)  
196     {  
197         //HERE GOES ANY CUSTOM IMPLEMENTATION IF THE CONDITION IS TRUE  
198     }  
199
```

Figure 1.29: “OR” operator

As seen in line 195, there is an if statement that contains two conditions separated by the keyword `||`, which is the equivalent to the **OR** operator in C#. In that case, the code inside the if block is executed only when at least one of the two conditions is satisfied: if the day of the week is Monday or Tuesday.

Otherwise, in the case where all the multiple conditions need to be true, the operator **AND** can be used to combine the expressions. The keyword for this operator is `&&` in C#, as shown in [Figure 1.30](#):

```
204     var dayOfWeekNow = DateTime.Now.DayOfWeek;  
205     var currentHour = DateTime.Now.Hour;  
206  
207  
208     if(dayOfWeekNow == DayOfWeek.Saturday && currentHour > 6)  
209     {  
210         //HERE GOES ANY CUSTOM IMPLEMENTATION IF THE CONDITION IS TRUE  
211     }  
212  
213
```

Figure 1.30: “AND” operator

In the example given, the condition to execute the block of code inside the if statement is satisfied only if the day of the week is Saturday and the current hour is greater than 6.

Unary operators

C# language has operators with the primary purpose of making operations using a single operand. They are timesavers to write specific routines such as incrementing a number, inverting a

Boolean value, decrementing a number, and many others. To add an extra number to an integer variable in C#, the operator `++` can be used, and it is possible to use the double minus operator to decrement the value of a variable, as seen in [Figure 1.31](#):

```
217     int numberExample = 25;
218
219     numberExample++; //variable has now the value of 26
220
221     numberExample--; //variable has now the value of 25 again
222
223     numberExample--; //variable has now the value of 24
224
```

Figure 1.31: Unary operators in C#

The above implementation is the same as if an arithmetic operator were used to change the value of the `numberExample` variable. In this case, it would be the same as: `numberExample = numberExample + 1`. Undoubtedly, the unary operator syntax is much more optimized regarding the amount of code needed to get similar results. However, in some instances, the unary operator may make the code difficult to read if too many of them are used in combination.

Ternary operators

Ternary operators are other ones that help us reduce the number of code lines in logical statements that can replace if statements in the cases where the result of a certain condition is the assignment of a variable. As shown in [Figure 1.32](#), there is a variable called `message` from the string type, as its value is a ternary expression using the operators `?` and `..`. If the condition is satisfied, a message `"Today is Monday"` is assigned to the variable; otherwise, the text `"Today is not Monday"` is assigned to the variable:

```
228
229     string message = DateTime.Now.DayOfWeek == DayOfWeek.Monday ?
230         "Today is Monday" : "Today is not Monday";
231
232
```

Figure 1.32: Ternary operators

It is not recommended to use conditions using ternary operators

when the condition is complex and may contain compound statements. In this case, too many conditions in the expression would make the code hard to read. Therefore, use the `if` statement instead and keep the logical operation explicitly exposed to facilitate maintenance and readability.

Compound assignment operators

Like ternary and unary operators, the bitwise type helps reduce the number of code lines in an implementation, making operations easier to make. Any arithmetic operator followed by the equal = operator means to the compile in C# language that the arithmetic operation before the equal operator should be applied to its variable value. [Figure 1.33](#) contains an example of the compound assignment operator for + (addition) and * (multiplication):

```
237     int firstExample = 25;
238     firstExample += 10; //25 + 10
239
240     int secondExample = 15;
241     secondExample *= 5; // 15 * 5
242
```

Figure 1.33: Compound assignment operators

Compound assignment operators should be used carefully as they may make a complex implementation hard to read and debug. Therefore, consider using explicit variables for math operations if they contain many parts and conditions.

If statement

C# language contains many operators for making different conditional statements in Boolean expressions. This indicates the importance of controlling the behavior of the software by implementing precise statements, even for complex conditions. The `if` statement is the most conditional operator in C# language, and it consists in executing or not a block of code based on one or more logical conditions. It is possible as well to specify a block of code to run only if a specific condition in the if statement is `false`, using the else clause, as seen in [Figure 1.34](#):

```

245     var currentMonth = DateTime.Now.Month;
246
247     if(currentMonth == 1)
248     {
249         Console.WriteLine("This is the month of January");
250     }
251     else
252     {
253         Console.WriteLine("This is NOT the month of January");
254     }
255
256     if (DateTime.Now.Day == 1)
257         Console.WriteLine("Today is the first day of the month");
258

```

Figure 1.34: If/else statement

In the example, a specific code block is executed if the current month corresponds to January, and another block is executed if the condition has a **false** result. This way of using **if** statements are particularly useful when two distinct things should be executed depending on the result: **true** or **false**. It follows the if-then-else pattern, and it is one of the most used features in C# in any application.

Suppose a logical requirement contains many sub-conditions inside of an if or else block, it is highly recommended to use other logical structures for conditions like the switch, which is explained in the next topic of this section. Depending on the complexity, many levels and sub-levels in nested conditions within a single implementation could turn the code hard to read and maintain, and errors might be introduced while changes are made in the implementation overtime.

Switch case

The **switch case** statement is suitable if a logical implementation contains multiple options to choose from; thereby, only one will have its code block executed. In other words, the option that matches a condition with a predefined value for comparison. The **switch case** is largely used in the cases where a value needs to be tested by many conditions or even if it has more than two or three conditions, as seen in [Figure 1.35](#):

```
262 switch (DateTime.Now.Month)
263 {
264     case 1:
265         Console.WriteLine("January");
266         break;
267     case 2:
268         Console.WriteLine("February");
269         break;
270     case 3:
271         Console.WriteLine("March");
272         break;
273     case 4:
274         Console.WriteLine("April");
275         break;
276     case 5:
277         Console.WriteLine("May");
278         break;
279     case 6:
280         Console.WriteLine("June");
281         break;
282     case 7:
283         Console.WriteLine("July");
284         break;
285     case 8:
286         Console.WriteLine("August");
287         break;
288     case 9:
289         Console.WriteLine("September");
290         break;
291     case 10:
292         Console.WriteLine("October");
293         break;
294     case 11:
295         Console.WriteLine("November");
296         break;
297     case 12:
298         Console.WriteLine("December");
299         break;
300 }
```

Figure 1.35: Switch-case statement

In this example, a value (month number) is tested against 12 options, one for each case in the instruction, and if one of the

conditions is satisfied, the program executes the block of code specified between the underlying case and break keywords. It is very useful to expose system requirements when the workflow for conditions could contain multiple options, and the routine that should be executed for each condition is slightly different. Using a **switch-case** statement for multiple options instead of multiple if clauses is a good practice in the C# programming language. If, in the example shown in [Figure 1.35](#), an if statement was used, the code would like the implementation in [Figure 1.36](#), which is not recommended:

```
303
304  if(DateTime.Now.Month == 1)
305  {
306      Console.WriteLine("January");
307  }
308  else
309  {
310      if(DateTime.Now.Month == 2)
311      {
312          Console.WriteLine("February");
313      }
314      else
315      {
316          if(DateTime.Now.Month == 3)
317          {
318              Console.WriteLine("March");
319          }
320          else
321          {
322              if(DateTime.Now.Month == 4)
323              {
324                  Console.WriteLine("April");
325              }
326          }
327      }
328  }
```

Figure 1.36: Nested if clauses

The implementation given in [Figure 1.36](#) works the same way as the previous in [Figure 1.35](#), but it contains a significantly greater number of lines and is much more complex to read. Therefore, it is recommended to use a **switch-case** statement when a value needs to be tested against multiple values.

Error handling

Every software existing in the market could have implementation errors, unexpected behaviors, and execution issues. There are many possible causes for software issues, such as conversion type errors, memory leak issues, data problems, missing null-checks, communication issues between the application and database, and server configuration problems, among many others. Each programming language has a particular way of handling those possible errors, and mapping those could notably contribute to more stable software.

The C# language contains native implementations to properly catch all the errors a program may have and show worthwhile and detailed information on the root cause of the exception. If the exception comes from any native .NET library, it also contains suggestions to fix the errors. Using third-party libraries and resources from other systems in any project is common. Integrating those third-party systems usually requires networking connections for communication, authentication, and protocols to function correctly. Considering many factors that could fail in these integrations, any application should have a reliable way to prevent the exceptions that could be raised in this context. The program must implement custom routines in case some issues occur in any part of the software.

Error handling in C# language uses three combined blocks of code, **try**, **catch**, and **finally** keywords, which allows us to specify proper custom actions if a software failure occurs. The implementation of exceptions can be applied on many levels of the application, and it is crucial to spend enough time to plan appropriately on the best strategy for error handling as part of project planning and architecture.

In software development, exceptions are all the errors that could happen in the application. The correct management of these exceptions make it possible to control the workflow of any custom implementation if any error occurs during the program execution, explicitly determining the application's behavior.

In [Figure 1.37](#), there is an example of exception handling implementation. In this context, the method receives a person's age

value and handles the possible exceptions throwing an exception to the system if the **age** is invalid:

```
331
332     int age = int.Parse(Console.ReadLine());
333
334     if (age < 0)
335     {
336         throw new ArgumentOutOfRangeException("The age must be greater than 0");
337     }
338
```

Figure 1.37: Throw exception

.NET libraries contain a lot of exception classes that inherit from the main **System.Exception** class. In the example given ([Figure 1.37](#)), the **Argument Out of Range Exception** is used, passing a custom message error to the system, referencing the cause of the issue, and enforcing the business requirement to not allow a negative number for the **age** value. An exception can be intentionally thrown in this case. Additionally, an unexcepted error might be thrown if the input given in line 332 is not an **integer** value.

Try-catch blocks should be used to manage the exception if the **age** value is invalid. Otherwise, the program's execution is interrupted because of unexpected behavior. The **try-catch** block can be specified as shown in [Figure 1.38](#):

```
332     try
333     {
334         int age = int.Parse(Console.ReadLine());
335
336         if (age < 0)
337         {
338             throw new ArgumentOutOfRangeException("The age must be greater than 0");
339         }
340     }
341     catch(ArgumentOutOfRangeException ex)
342     {
343         Console.WriteLine($"A valid value for the age need to be provided. Error details: {ex.Message}");
344     }
345     catch(Exception ex)
346     {
347         Console.WriteLine($"Unexpected error. Details: {ex.Message}");
348     }
349
```

Figure 1.38: try-catch block

As seen in [Figure 1.38](#), the **age** variable is assigned inside the **try** block. If an exception occurs, the system executes the underlying **catch** block according to the exception type. If the error has a known cause handled by custom exception handling, more detail on the error is given because the **Out Of Range** exception is explicitly thrown because of business logic requirements. If any other error

occurs, the second **catch** block is executed.

Exception handling is one of the most important parts of any project of software because it is closely related to how the application handles user experience if some error occurs, and it is directly related to software reliability as well. Therefore, all the possible exceptions must be mapped in the development and design phases of the development life cycle, and a custom action should be taken respecting business decisions and a good software architecture approach.

When an application uses external resources or has integrations with third-party systems, a routine usually needs to be implemented to prevent disposal issues of objects in the memory. For instance, if there is an error to record something into a database, it is recommended to close the connection with the same database, regardless of whether the process succeeds. To handle this situation correctly, it is possible to use the **finally** block, which is executed in all the cases in the code workflow, regardless of if the catch block is executed or not, as seen in [Figure 1.39](#):

The image shows a code editor with a dark background and light-colored text. The code is in C# and demonstrates exception handling with a try-catch-finally block. Line numbers 332 through 354 are visible on the left. The code starts with a try block containing a call to int.Parse(Console.ReadLine()) and a conditional check for age < 0. If the condition is true, an ArgumentOutOfRangeException is thrown. After the try block, there is a catch block for ArgumentOutOfRangeException that writes an error message. Then, there is another catch block for a general Exception. Finally, a finally block contains a call to CloseDatabaseConnection().

```
332 try
333 {
334     int age = int.Parse(Console.ReadLine());
335
336     if (age < 0)
337     {
338         throw new ArgumentOutOfRangeException("The age must be greater than 0");
339     }
340
341     SavePerson();
342
343 catch(ArgumentOutOfRangeException ex)
344 {
345     Console.WriteLine($"A valid value for the age need to be provided. Error details: {ex.Message}");
346
347 catch(Exception ex)
348 {
349     Console.WriteLine($"Unexpected error. Details: {ex.Message}");
350 }
351 finally
352 {
353     CloseDatabaseConnection();
354 }
```

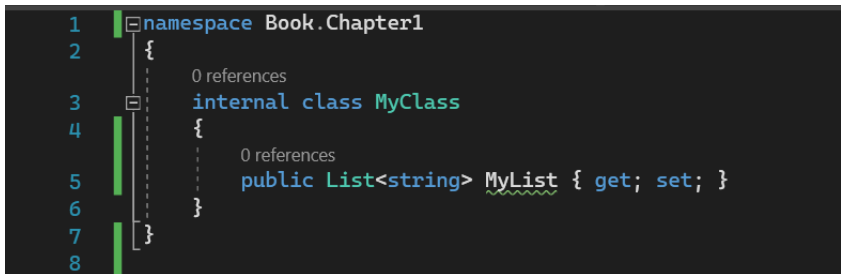
Figure 1.39: Finally block

In [Figure 1.39](#), it is possible to see that, after validating if the **age** is valid or not inside the **try** block, the application calls the **SavePerson** method, which in real scenarios would make an actual database operation that requires an open connection and depends on network availability to establish communication. Even if the operations fail, the system always closes the connection with the database, as shown in lines 351 and 354. The **finally** block code

is executed in all the cases, respecting the order of **try** and **catch** blocks order of execution.

Namespaces

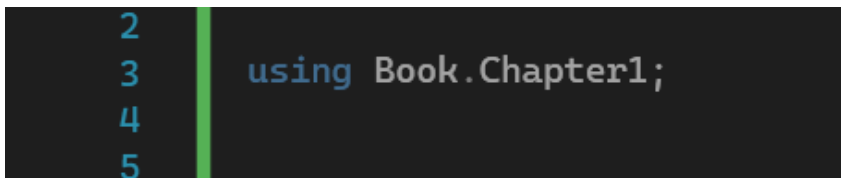
A namespace in C# language is a feature that allows us to separate and keep a set of classes, structure, and other implementations based on logical criteria. Every class is under a specific namespace in C#; the ones we create ourselves or the ones provided by the .NET platform. To define a namespace, specify a path separated by a period before the name of the class or struct, as seen in [Figure 1.40](#):



```
1 namespace Book.Chapter1
2 {
3     0 references
4     internal class MyClass
5     {
6         0 references
7         public List<string> MyList { get; set; }
8     }
```

Figure 1.40: Namespace definition

In the example given, the **MyClass** class is under the namespace **Book.Chapter1**, therefore, it would be logical to keep other related classes under the same namespace. In the cases where the class needs to be used within another namespace, a **using** directive can be used to get visibility of the class, as shown in [Figure 1.41](#):



```
2
3 using Book.Chapter1;
4
5
```

Figure 1.41: using namespaces

Once a namespace is referenced in any C# file, all the classes and structs under this namespace are available to use in the context of the same file, depending on the visibility of the class, if that is marked as **internal**, **private**, or **public**. Using namespaces allows to organize complex projects to keep the classes and structs under a similar box.

New features in C# 10 and 11

.NET 7 brought a relevant number of general improvements in the .NET platform, and it is applicable to enhancements for C# language in version 11. In this section, you will walk through the most relevant improvements and new features introduced in C# language since C# 10 and the new features expected in C# 11.

Global using directives

Since C# 10, you can specify any using directive as global, meaning there is no need to declare individual using directives across all the files that need that reference. There are two ways to use global using directives: one is to use the global keyword before a regular using directive, as seen in [Figure 1.42](#):

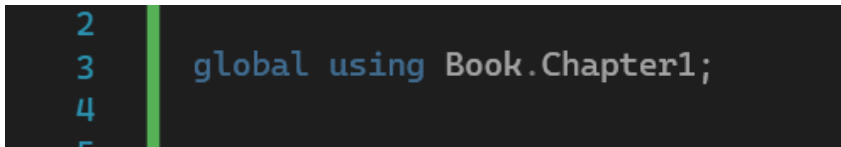
A code editor snippet showing a C# file with line numbers 2, 3, 4, and 5 on the left. On line 3, the code `global using Book.Chapter1;` is written in a light blue font on a dark background.

Figure 1.42: Global using directive

Another way to achieve the same objective is to edit the underlying project file (`.csproj`), specifying the `Using` tag in the project configuration, as seen in [Figure 1.43](#):

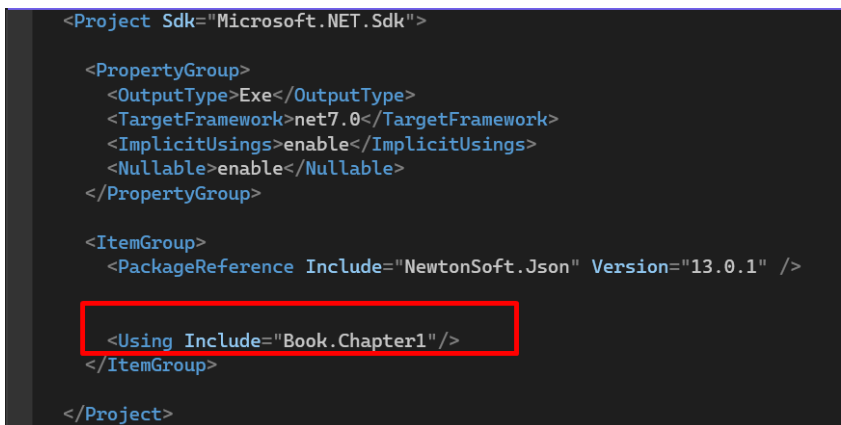
A screenshot of an XML project file (.csproj) in a code editor. The XML structure includes <Project>, <PropertyGroup>, <ItemGroup>, and </Project> tags. Inside the <ItemGroup>, there is a <PackageReference> tag and a <Using Include="Book.Chapter1"/> tag. The <Using> tag is highlighted with a red rectangular box.

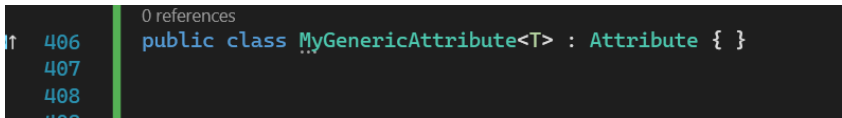
Figure 1.43: Global using directive in .csproj

Both ways have the same result, but in terms of maintenance and

readability, the specification of global using directives in the project file may represent a more explicit way to centralize all the global directives.

Generic Attributes

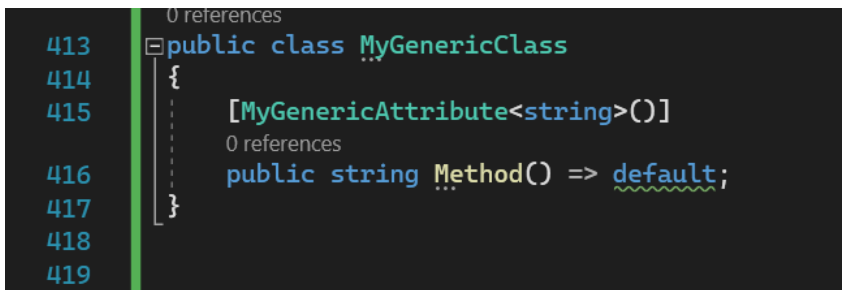
It is possible to create a generic class in C# inherited from the **System.Attribute** class. It simplifies the syntax if compared to previous versions of the C# language. For instance, you have to create a **GenericAttribute** class, as seen in [Figure 1.44](#):



```
0 references
406 public class MyGenericAttribute<T> : Attribute { }
407
408
409
```

Figure 1.44: Generic Attribute class

After creating the **Generic Attribute** class, you can use it as an annotation above class properties, as seen in [Figure 1.45](#):



```
0 references
413 public class MyGenericClass
414 {
415     [MyGenericAttribute<string>()]
0 references
416     public string Method() => default;
417 }
418
419
```

Figure 1.45: Generic attribute usage

The type argument in the attribute is mandatory in that case, which means that restrictions are in place to avoid conversion errors, similar to when the **typeof** operator is applied to verify the type of a specific object.

List patterns

List patterns allow us to match item sequences in a list or array. Therefore, it is not mandatory to loop through all the elements in a list anymore to check if a sequence of elements is correct or if a specific sub-list of elements is presented on the list, as seen in [Figure 1.46](#):

```

8
7    int [] myList = new int [] { 1, 2, 3 };
8
9    Console.WriteLine(myList is [1, 2, 3]); //true
10   Console.WriteLine(myList is [1]); //false
11

```

Figure 1.46: List pattern

If the length of elements in the arrays used for comparison does not match, the result always returns false for the condition. Therefore, the elements should have a strict and exact match.

Raw string Literals

C# 11 introduced a new way to format string literals. The use of three double-quotes(“””) characters makes it possible to include texts in the **string** with whitespace, new lines, extra quotes as part of the content, and other characters that before were not allowed without special formatting. The new feature is shown in [Figure 1.47](#):

```

16
17   string rawMessage = """
18       This is a message with new line, whitespace
19       and other characters,
20       including "quotes" as part of the content
21
22   """;
23
24   Console.WriteLine(rawMessage);
25
26

```

Figure 1.47: Raw string literals

C# 11 contains many other new features, such as extended **nameof** scope, auto-default struct, improved method group conversion to delegate, numeric **IntPtr** and **UIntPtr**, newlines in string interpolations, and many others. This book demonstrates some of these new features in the Design Pattern examples.

New debugger features in Visual Studio

As Visual Studio 2022 is being used across the code samples of this book, it is worthwhile to mention and highlight the debugger features introduced in this new Visual Studio version to facilitate

the experience of building software using the C# language.

Temporary breakpoints

Visual Studio allows us to create breakpoints to help in the debugging process in the cases where the execution of the application needs to be verified slowly and step by step. Now, it is possible to specify temporary breakpoints in Visual Studio, as seen in [Figure 1.48](#):

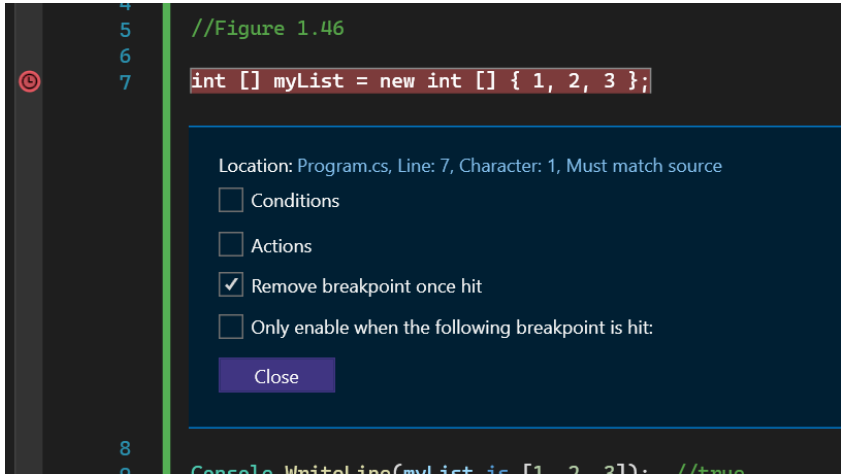


Figure 1.48: Temporary breakpoints

In the example given, a breakpoint is specified in line 7, and the **Remove breakpoint once hit** option is checked, which means that once the debug process reaches the respective line once, the breakpoint is automatically removed from Visual Studio after that.

Breakpoint hover glyph

Not all points in the code can be reached by a breakpoint. To help in the determination of which points are available to use breakpoints in the code, Visual Studio provides a navigation cursor in the breakpoint sidebar to help the user experience, as seen in [Figure 1.49](#):

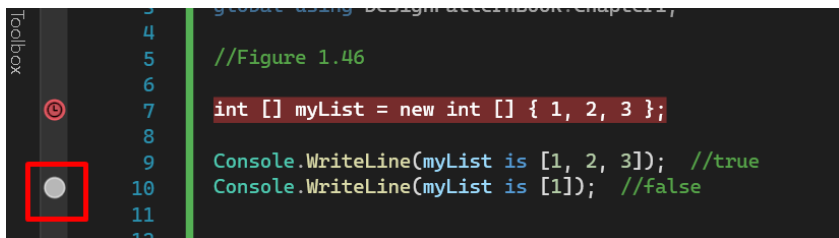


Figure 1.49: Breakpoint hover glyph

Small details in terms of usability make a difference in the developer experience using Visual Studio.

Conclusion

Before you get started with Design Patterns using C# 11.0 and .NET 7, you must get familiar with the most basic concepts of C# language, such as error handling, variables, loop, operators, and conditional statements. The fundamental programming language knowledge is essential to apply correctly object-oriented concepts and implement advanced architectures using Design Patterns. In the next chapter, you will be learning concepts on .NET Core.

Points to remember

- To develop .NET applications based on .NET 7, you must install the latest version of Visual Studio or Visual Studio Code
- The C# language contains features to handle exceptions and system errors in the software execution.
- To start with C#, you must be familiar with the variable types, loop statements, operators, and error handling.

Multiple Choice Questions

1. Which option contains only relational operators:

- A. `=`, `!=`, `>`, `<`
- B. `!=`, `>`, `<`, `if`
- C. `switch`, `for`, `foreach`, `equal`
- D. None of these

2. Which type of variable is used to store true and false values:

- A. string
- B. int
- C. bool
- D. decimal

3. What is the value of the variable “isValid” in the following code:

```
int age = 25;  
var isValid = age > 18;
```

- A. false
- B. true
- C. 25
- D. 18

4. After the following code, what is the final value of the number variable?

```
int number = 78;  
number++;
```

- A. 78
- B. 77
- C. 80
- D. 79

5. Imagine a routine in your code that should connect to a database. Considering a networking issue could happen, in which block of code is recommendable to close the connection with the database:

- A. Inside the try block
- B. Inside the catch block
- C. Inside the finally block

Answers

--	--

	A	
	B	
	C	
	D	
	E	
	F	
	G	
	H	
	I	
	J	
	K	
	L	
	M	
	N	
	O	
	P	
	Q	
	R	
	S	
	T	
	U	
	V	
	W	
	X	
	Y	
	Z	

Questions

1. What are the available loop statements in C#?
2. What is the best alternative to implement a logical statement that could have more than three conditions?
3. What do the try-catch blocks for?

Key terms

- **Exceptions:** Errors that could happen in software execution.
- **Operators:** The C# language contains many operators that allow to make arithmetic operations and simplify iterations and testing conditions

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

.NET Fundamentals

Introduction

With this chapter, we are starting our journey into the essential aspects of .NET that a developer needs to get started with the platform, including multi-platform concepts, the most relevant new features and project templates introduced in .NET 6 and .NET 7, performance enhancements, and new possibilities for creating minimal APIs.

You will learn in this chapter how to create various projects using the default templates available in Visual Studio, such as Console and Desktop applications, Blazor, ASP.NET Core WEB API, and an introduction to the MAUI project.

Getting familiar with the basic concepts of .NET will allow you to understand how to apply different types of projects in distinct scenarios, helping you develop your career.

Structure

In this chapter, we will discuss the following topics:

- Understanding of multi-platform concepts

- .NET for cloud development
- New features in .NET 6 and .NET 7
- Performance enhancements on .NET 6 and .NET 7

Objectives

Upon completing this unit, you will acquire several essential capabilities. You'll become skilled at crafting minimal APIs within the .NET framework, while also gaining a solid grasp of cloud development principles using .NET. Moreover, you'll refine your expertise in enhancing existing codebases for improved performance. Lastly, you'll develop insights into the evolving trends of the .NET platform, equipping you to anticipate its forthcoming advancements.

Multi-platform concepts

With the beginning of the cloud-based applications in the last decade, the market competition changed significantly from a battle for a software license to a cloud system infrastructure service to support the high demand for scalability, big data, globalization, and open-source project needs. Therefore, the interest in profitability is no longer focused on operating systems but on infrastructure reliability as many companies are changing their business model to provide solutions based on the **Software As a Service (SaaS)** model.

Nonetheless, big tech companies are still realizing and creating new versions of operating systems, the attention shifted completely to following an open-source tech culture, with common sense by all those big companies that any modern application must be able to run in Linux, macOS, and Windows platforms, without having to maintain a separated codebase for each of them.

Furthermore, with the rise of the **Internet of Things (IoT)**, the types of devices into which applications can be executed are more diverse than ever, making the software development process much more complex. Not long ago, every Web developer's biggest concert successfully ran the same application in multiple browsers (Internet Explorer, Firefox, Chrome, etc.) with reasonable quality. In this context, Web applications could be accessed from various operating

systems, and the applications were hosted on a server, where the specs and configurations are pretty under control. But, this relatively controlled scenario rapidly moved to another more complex one, based on a cross-platform environment to have more application compatibility in multiple operating systems, decrease infrastructure costs, and deliver applications faster to a more significant number of users on a global scale using microservice architecture.

To meet these new requirements for modern software development, Microsoft provided improvements on the .NET platform since the .NET Core 1.0 version, having a pretty mature framework for cross-platform development since .NET 5, and relevant enhancements in the subsequent versions until .NET 7. Therefore, .NET 7 is a rich and powerful technology that allows us to build modern applications that can be executed literally *everywhere*, with the bonus of having a higher performance than previous .NET versions.

At the very beginning of the C# language within the .NET Framework, Microsoft technologies and tools for software development were primarily done to be supported only on the Windows platform, which represented an explicit limitation for the adoption of .NET to a more significant portion of the market as the development of Desktop applications were popular at the time and this type of project required to be written twice or even more times redundantly, one codebase per operating systems, increasing the complexity and implicit costs of software projects in general.

In that sense, the .NET platform changed the game since the revolutionary change to be more open-source focused since .NET Core 1.0, bringing together cross-platform capabilities. Building an application using the latest .NET version (.NET 7) allows you to write a single application that can be hosted on any platform, in multiple types of devices, distinct cloud providers, and be consumed by plenty of Web APIs. This revolution placed .NET in a prominent position in the market, with individuals, companies, technical communities, and academic institutions able to contribute to the platform's evolution through open-source project models.

.NET for cloud development

Modern software development involves knowledge of cloud services and architecture patterns suitable for distributed systems, allowing developers to create applications that are scalable, reliable, and cost-effective in terms of infrastructure. The .NET platform offers a wide range of libraries and packages to facilitate the integration with Azure, the powerful cloud platform provided by Microsoft.

The numbers of services available on Azure constantly increases, including services such as: Virtual Machines, Cloud Storage, Serverless functions, relational and no-SQL databases, Web servers, IoT, Mixed Reality, Artificial Intelligence, Networking, Active Directory, DevOps, Blockchain, Machine Learning and much more.

For each of these services, Microsoft provides SDKs and libraries for C# and other languages, facilitating the integration of business applications with multiple cloud services. The majority of resources on Azure has the possibility of using traditional REST APIs for integration, but the use of libraries as middleware definitely speeds up the development process, exposing the APIs in a way developers can take a huge advantage in terms of reusability and readability.

Azure storage accounts

Most applications need to store images, static files and content from different formats and allow users to upload their own files to the server. Similar requirement is applicable for web, mobile and desktop applications, including background services that need to process a huge amount of files. There are many ways of storing these files, such as:

- Store these files in a folder within the application server
- Have a dedicated machine to store the files
- Store the files in folders among the application files
- Store them in a database as blobs
- Use a storage system to store the files outside of the application context

All these options have pros and cons depending on the context of the technical and non-functional requirements that an application must have, but in general, there are good practices in terms of file upload that are recommended to follow in any project. One of these

good practices involves the separation of user content files from the application server into a separate external storage service, which allows a more efficient way to provide files (bandwidth) and has high capabilities for data replication, security policies, and automatic backup tools.

Considering these practices, Azure Storage Account represents a great alternative in terms of scalability, security, performance and cost management; being possible to use this cloud resource with .NET applications as Microsoft provided packages for C# language to speed up the development process, even though it is possible to do any integration with Azure services using traditional REST APIs as well.

To integrate C# applications with Azure Storage Accounts, you need to install the `Azure.Storage.Blobs` package provided by Microsoft. To test the package, create a new Console Application using Visual Studio 2022 or Visual Studio Code and install the underlying package, as seen in *Figure 2.1*:

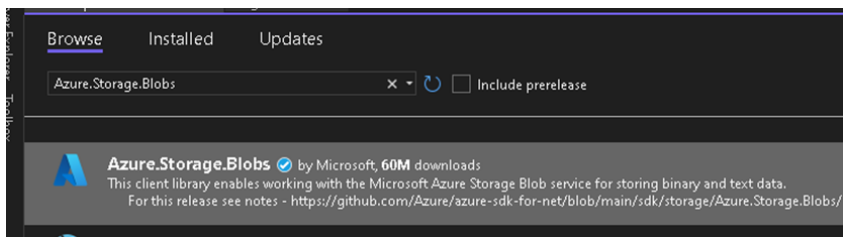


Figure 2.1: Azure Storage Blobs package

Within the `Program.cs` file, you would need to import a reference to the package and specify the authentication information regarding your storage account, such as `connectionstring` and `container name`, as seen in *Figure 2.2*:

```

1  using Azure.Storage.Blobs;
2
3  namespace Chapter2
4  {
5      internal class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Uploading file");
10
11              string azureAccountConnectionString = "YOUR CONNECTIONSTRING";
12              string containerName = "upload";
13              BlobContainerClient container =
14                  new BlobContainerClient(azureAccountConnectionString, containerName);
15
16              container.CreateIfNotExists();
17          }
18      }
19  }
20

```

Figure 2.2: Authentication information for Azure Storage

Note in [Figure 2.2](#) that the first code line contains the reference to the **Azure.Storage.Blobs** package. Furthermore, the eleventh line contains a variable to store the **connectionstring** for the container, and line 12 has the container's name created under the underlying storage account. You can obtain the connectionstring for your storage account under the **Access Keys** section on Azure Portal, as shown in [Figure 2.3](#):

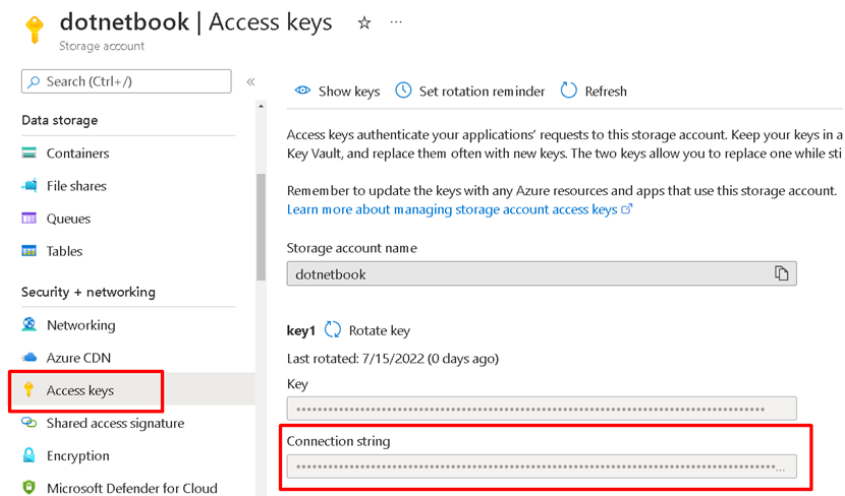


Figure 2.3: Connectionstring for Azure Storage Account

Once you get the **connectionstring** you can replace the

azureAccountConnectionString value. In real scenarios, this type of sensitive information should not be part of the codebase but a protected environment variable on the application server. Therefore, the hard-coded connectionstring is just for this sample code.

Looking at the continuation of the given code sample, note that lines 12 and 13 contain the creation of a **BlobContainerClient** instance, which is part of the package installed and allows the application to establish communication with the Azure Storage Account passing the proper connectionstring and container information as highlighted in [Figure 2.4](#):

```
BlobContainerClient container =  
    new BlobContainerClient(azureAccountConnectionString, containerName);  
  
container.CreateIfNotExists();
```

Figure 2.4: Blob container client object

In order to start uploading files to a specific container, it is important to make sure the container gets created if that does not exist yet. Therefore the method **container.CreateIfNotExists()** is called in this sample. In the context of this chapter, the container is called **upload**.

Once you have created the underlying container, the next step would be to upload an actual file to the container. To achieve that, you can create a text file in your local machine, create a **BlobClient** object referencing the path that the file should be placed in the container, and finally call the **Upload** method, passing the file stream of your sample file, as shown in [Figure 2.5](#):

```
string blobName = "sample.txt";  
string filePath = $"myfiles/{blobName}" ;  
  
BlobClient blob = container.GetBlobClient(filePath);  
  
blob.Upload(File.OpenRead("C:\\Users\\Alexandre Malavasi\\Desktop\\sample.txt"));  
Console.WriteLine("Upload done successfully");
```

Figure 2.5: File upload to the Azure Storage container

In the context of this chapter, a file called `sample.txt` was created manually and placed on the Desktop, and the underlying path was passed as a parameter of the `OpenRead` method to get the underlying file stream. This code specifies that the `sample.txt` file will be uploaded into the specified container under the path `myfiles/sample.txt`. [Figure 2.6](#) contains the entire code pulled together for the example:

```
1 using Azure.Storage.Blobs;
2
3 namespace Chapter2
4 {
5     0 references
6     internal class Program
7     {
8         0 references
9         static void Main(string[] args)
10         {
11             Console.WriteLine("Uploading file");
12
13             string azureAccountConnectionString = "YOUR CONNECTIONSTRING";
14             string containerName = "upload";
15             BlobContainerClient container =
16                 new BlobContainerClient(azureAccountConnectionString, containerName);
17
18             container.CreateIfNotExists();
19
20             string blobName = "sample.txt";
21             string filePath = $"myfiles/{blobName}";
22
23             BlobClient blob = container.GetBlobClient(filePath);
24             blob.Upload(File.OpenRead("C:\\Users\\Alexandre Malavasi\\Desktop\\sample.txt"));
25
26             Console.WriteLine("Upload done successfully");
27         }
28     }
29 }
```

Figure 2.6: Code sample for the file upload

After running the Console application, the message regarding the successful upload will be displayed if all the parameters are correct, such as `connectionstring`, `file path`, and others, as seen in [Figure 2.7](#):



```
Microsoft Visual Studio Debug Console
Uploading file
Upload done successfully

C:\Users\Alexandre Malavasi\source\repos\BpbDotNetBook\bin\Debug\net6.0\BpbDotNetBook.exe (process 5320) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 2.7: Message for successful upload

After running the application, you will be able to access and see on

Azure Portal the underlying folder and file uploaded into the corresponding Storage Account, as seen in [Figure 2.8](#):

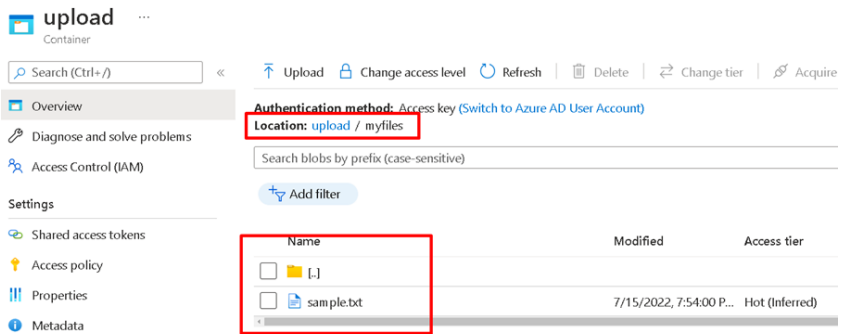


Figure 2.8: Folder and file on the Storage container

The package to manipulate blobs in the storage account provided by Microsoft is quite powerful in terms of mirroring all the capabilities that are presented in this type of resource on Azure, allowing to upload, download, delete, get a list of files, change metadata, control security and do any other operations for blobs in general.

Furthermore, Storage Accounts on Azure have many other extra capabilities, such as Tables, Queues, Networking configuration, Azure CDN, Static websites, and much more, representing a great tool to extend the enterprise application's capabilities.

Azure Function

Azure Function is based on the serverless computing model that allows us to execute code using the events and triggers approach without having to take care of how the infrastructure for the host application is configured, being a cost-effective option for scalable services.

Imagine you have a specific service that needs to be executed sporadically or only when a particular event happens. In this case, it does not make sense to keep a regular application server available 24 hours and seven days a week, increasing costs and overpricing the infrastructure for your service. It would be suitable to have a specific service that is online only when a particular event happens, and once the process is finished, the infrastructure would be turned

off automatically.

Azure Function is a perfect alternative for this kind of operation, as it allows us to create function for short-lived processes, billing the resource per proportional use. Therefore, depending on the configuration of an Azure function app, it would support millions of requests per month for a reasonable price.

To start using Azure Functions, a Function App must be created on Azure, which can host multiple functions together. If you got the Azure Portal within the **Create a Resource** option, you can create a **Function App**, as seen in *Figure 2.9*:

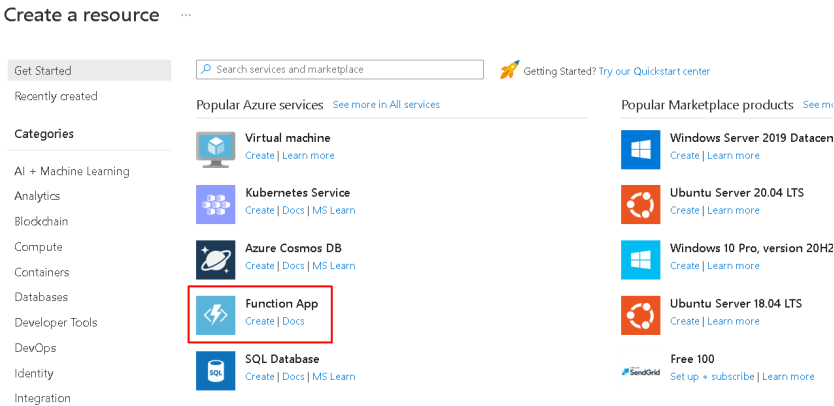


Figure 2.9: Function app creation

Similar to other resource types on Azure, it is mandatory to specify a unique name for your resource. It applies not only to your account, but to all the resources presented on Azure as Function apps use the domain **azurewebsites.net** by default, and the name of your App can not conflict with other existing Function Apps URL. You have to specify a subscription and a resource group as well, as seen in *Figure 2.10*:

Create Function App ...

Basics Hosting Networking Monitoring Tags Review + create

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Resource Group * ⓘ [Create new](#)

Instance Details

Function App name * ☒ .azurewebsites.net

Figure 2.10: Function app name

Azure functions support code implementation using .NET, Node.js, Python, Java, and Powershell script. In the context of this chapter, it was chosen .NET as the primary stack development, with the version six of the platform, as seen in [Figure 2.11](#):

Publish * ☒ Code ☐ Docker Container

Runtime stack *

Version *

Region *

Figure 2.11: Function app stack

The rest of the basic configuration is regarding **Operating System** and **Plan**, which directly influences the pricing model to keep the infrastructure of the the app itself and to execute functions. In this example, the Linux operating system was chosen as .NET is multi-platform since .NET Core 1.0 version. The **Consumption (Serverless)** plan type was also chosen, as seen in [Figure 2.12](#):

Operating system

The Operating System has been recommended for you based on your selection of runtime stack.

Operating System *

☒ Linux ☐ Windows

Plan

The plan you choose dictates how your app scales, what features are enabled, and how it is priced. [Learn more](#)

Plan type * ⓘ

Consumption (Serverless) ▼

Figure 2.12: Operating system and plan type

After confirming the basic configuration, you are redirected to the preview screen, where you have the opportunity to check each relevant aspect of your app before confirming its actual creation. The majority of configurations under the basic ones cannot be changed after their creation; therefore, it is essential to review the values specified in the last screen before confirming the creation of the resource, as seen in [Figure 2.13](#):

Create Function App ...

Basics Hosting Networking Monitoring Tags **Review + create**

Summary



Details

Subscription

Resource Group bpbdotnetbook_group

Name bpbdotnetbook

Runtime stack .NET 6

Hosting

Storage (New)

Storage account bpbdotnetbookgroup8c83

Plan (New)

Plan type Consumption (Serverless)

Name ASP-bpbdotnetbookgroup-b841

Operating System Linux

Region Central US

SKU Dynamic

Figure 2.13: Function app review screen

If the deployment succeeds, you should be able to see the details of all the resources that were created, including a storage account, sites and other components, as shown in [Figure 2.14](#):

✔ Your deployment is complete



Deployment name: Microsoft.Web-FunctionApp-Portal-edfc4b5a-bb74

Start time: 7/16/2022, 10:12:26 AM

Subscription: Assinatura do Visual Studio Enterprise

Correlation ID: f9678212-0bfe-4e22-bd05-af2348a1065f

Resource group: bpbdotnetbook_group



Deployment details (Download)

Resource	Type	Status
✔ bpbdotnetbook	Microsoft.Web/sites	OK
✔ bpbdotnetbookgroup8c83	Microsoft.Storage/storageAccounts	OK
✔ ASP-bpbdotnetbookgroup-b841	Microsoft.Web/serverfarms	OK
✔ bpbdotnetbookgroup8c83	Microsoft.Storage/storageAccounts	OK
✔ bpbdotnetbook	microsoft.insights/components	OK
✔ bpbdotnetbook	microsoft.insights/components	OK

Figure 2.14: Function app deployment details

After the Function App is generated, you can manually create functions under the new resource, as presented in [Figure 2.15](#):

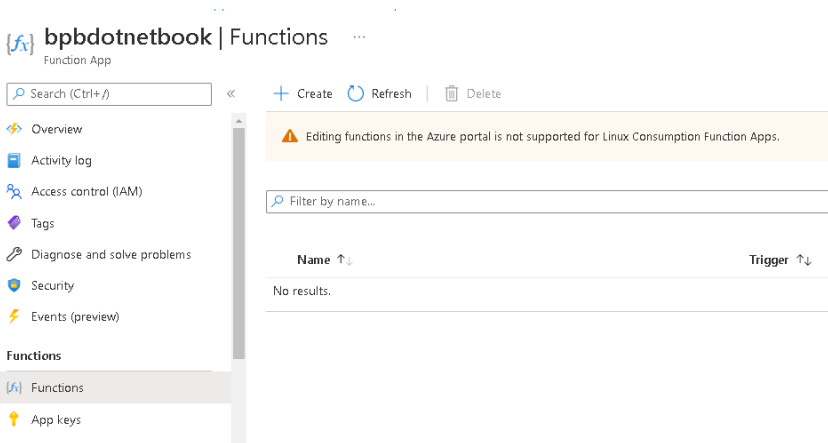


Figure 2.15: Functions on Azure Portal

Considering Azure functions host a portion of code in a small scope to be executed, it is possible to develop the functions locally using Visual Studio or Visual Studio Code. In this case, the underlying Azure development workload needs be installed as part of the Visual Studio installation, as highlighted in [Figure 2.16](#):

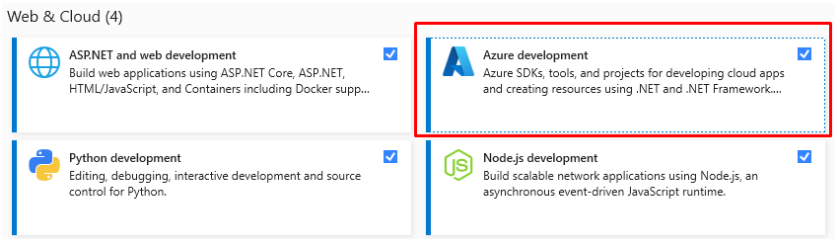


Figure 2.16: Azure development workload

After the installation of the workload, all the relevant project templates for Azure development are available in Visual Studio when you start a new project from scratch. To experience local development for Azure Functions, you would need to create the underlying project in Visual Studio, as shown in [Figure 2.17](#):

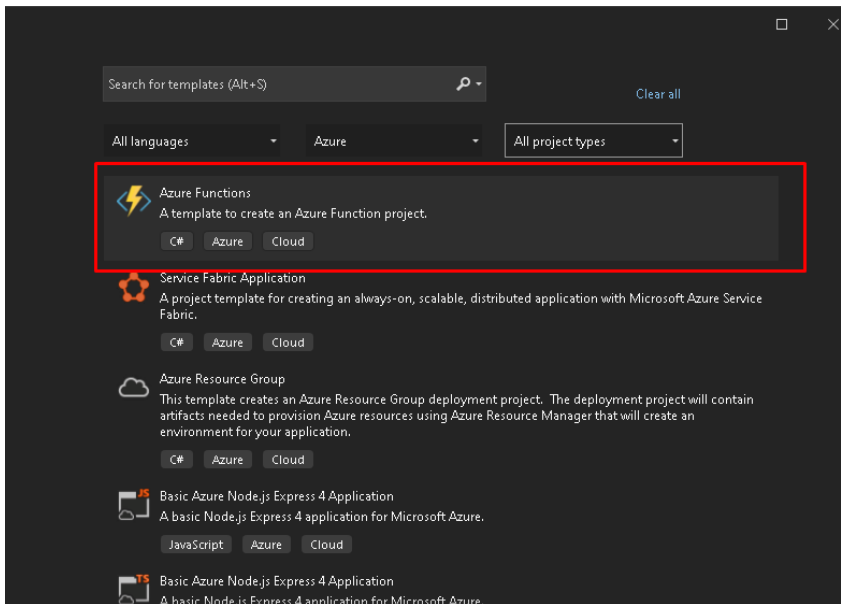


Figure 2.17: Azure function project in Visual Studio

After choosing the underlying project type and giving the project a name, you must choose the .NET version, the trigger type, and the authorization level. In the samples of this chapter, the .NET 6 version was chosen, combined with HTTP trigger and anonymous access, as seen in [Figure 2.18](#):

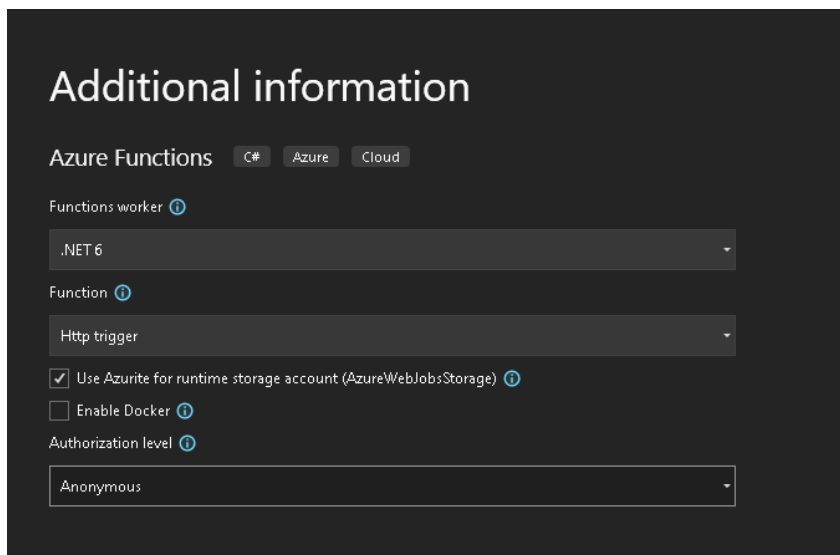


Figure 2.18: Azure function project configuration

The HTTP trigger option means that the function will be executed once the Web endpoint for the function is called by a **GET**, **POST** or **PUT** request, depending on the configuration for the function app itself. Regarding the authorization level, in production environments, another type of authentication should be used for security reasons, but for testing and studying purposes, anonymous access is used in this context of this chapter.

The default project template for HTTP trigger type creates a static class called **Function1**, a local settings file, and information on the host, as seen in [Figure 2.19](#):

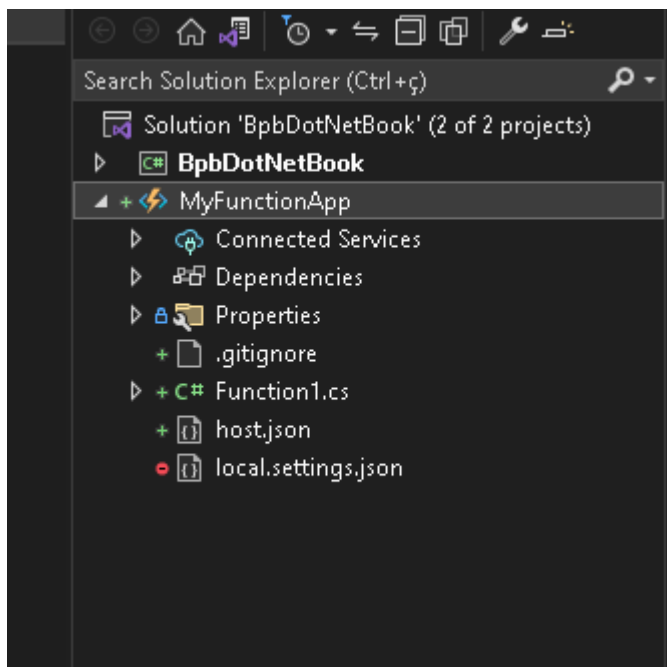


Figure 2.19: Solution Explorer for Function Apps

If you click at the **Function1** file, you will be able to see the default function created as an example by the project template. It contains the configuration for the authentication level, allowed HTTP Verbs, and routing specification. Considering the standard workflow for any application based on HTTP requests, it is possible to say that an Azure Function App based on HTTP trigger is quite similar to a standard RESTful Web API, considering all the following characteristics:

- Authentication and authorization configuration
- Multiple methods
- Multiple types of action results
- HTTP verb and route configuration
- Dependency injection

Therefore, if you are familiar with ASP.NET Core Web API projects, understanding Azure Functions will be facilitated for specific trigger types. After creating a project with the default project template, the **Function1** class looks like the representation in [Figure 2.20](#):


```

11 namespace MyFunctionApp
12 {
13     0 references
14     public static class Function1
15     {
16         [FunctionName("Function1")]
17         0 references
18         public static async Task<ActionResult> Run(
19             [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpRequest req,
20             ILogger log)
21         {
22             log.LogInformation("Developing my first Azure Function.");
23
24             string name = req.Query["name"];
25
26             string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
27             dynamic data = JsonConvert.DeserializeObject(requestBody);
28             name = name ?? data?.name;
29
30             string responseMessage = string.IsNullOrEmpty(name)
31                 ? "This HTTP triggered function executed successfully. Pass a name in the"
32                 : $"Hello, {name}. This HTTP triggered function executed successfully.";
33
34             return new OkObjectResult(responseMessage);
35         }
36     }
37 }

```

Figure 2.20: Function1 code

GET and **POST** HTTP verbs are allowed in the given function, and the authorization level is set to Anonymous on lines 16 and 17. The method gets the value of a **name** parameter passed as part of the URL request and returns a string response with the message:

"Hello. This HTTP triggered function executed successfully." If you run the application using Visual Studio, a simulated Azure Function App infrastructure will be executed locally, and a localhost endpoint will be provided, as seen in [Figure 2.21](#):

```

C:\Users\Alexandre Malavasi\AppData\Local\AzureFunctionsTools\Releases\4.10.3\cli_x64\func

Azure Functions Core Tools
Core Tools Version:          4.0.3971 Commit hash: d0775d487c93ebd49e9
Function Runtime Version: 4.0.1.16815

[2022-07-16T10:30:15.253Z] Found C:\Users\Alexandre Malavasi\source
user secrets file configuration.

Functions:

    Function1: [GET,POST] http://localhost:7071/api/Function1

For detailed output, run func with --verbose flag.
[2022-07-16T10:30:30.736Z] Host lock lease acquired by instance ID

```

Figure 2.21: Azure function local execution

The endpoint <http://localhost:7071/api/Function1> was created, and it can be executed directly in the browser as a standard Web API, as seen in [Figure 2.22](#):

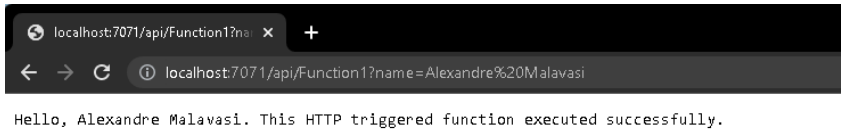


Figure 2.22: Azure function browser execution

Please note in the URL bar that the name parameter was passed as part of the request with the value **Alexandre Malavasi**. Considering the configuration of the function code for the **Function1** method, the correct output is displayed, as highlighted in [Figure 2.23](#):

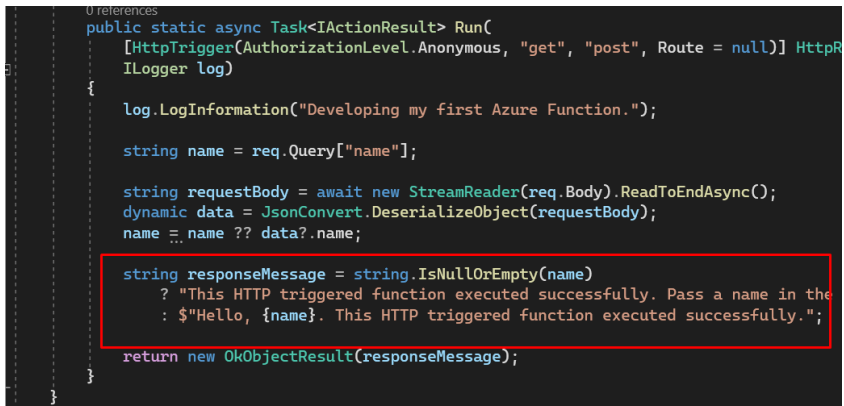


Figure 2.23: Azure function output

To test the execution of the app developed locally, it is possible to deploy the same code to the an actual Azure Function App that can be created via Azure Portal. For studying purposes, you can use the **Publish Selection** option under the Build menu option in Visual Studio, as shown in [Figure 2.24](#):

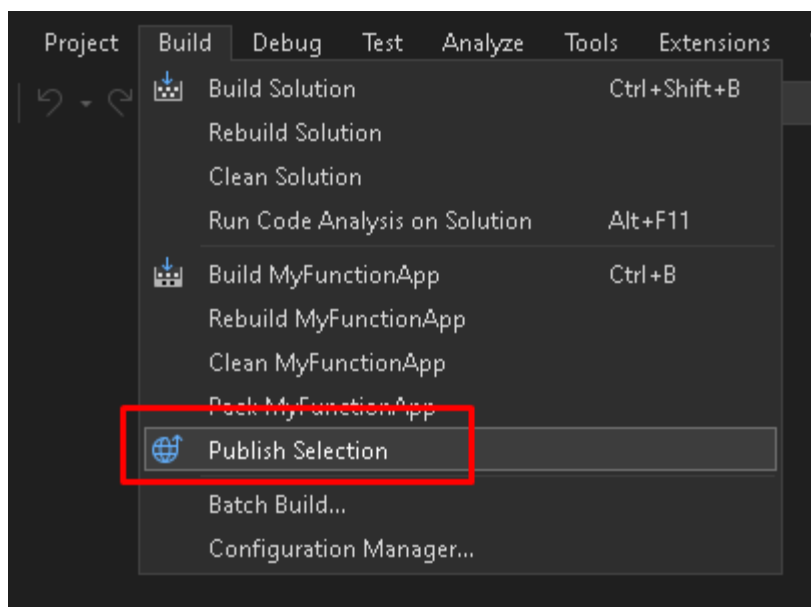


Figure 2.24: Publish option

After choosing this options, four alternatives are displayed, including the Azure deployment option, which is the correct option in the context of this chapter, as the intention is to deploy the Azure Function developed locally to the cloud. Select the corresponding option, as seen in [Figure 2.25](#):

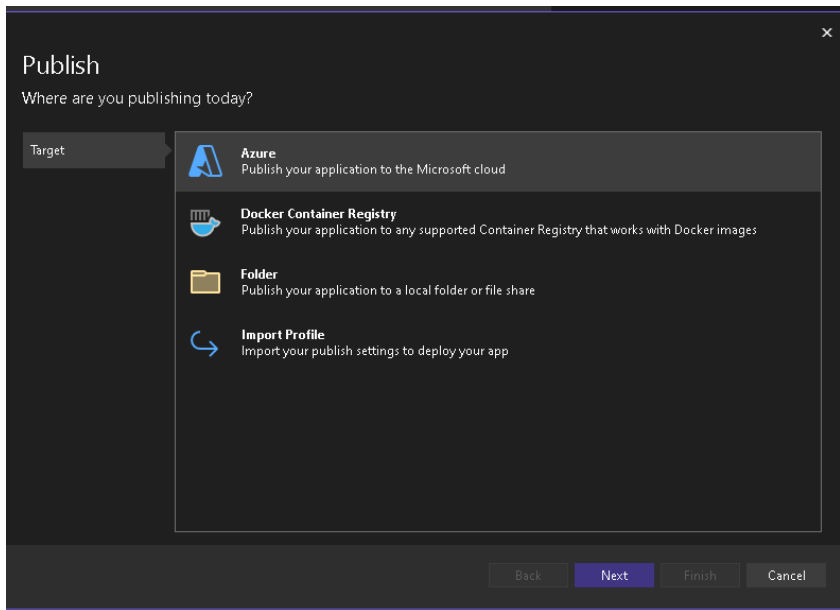


Figure 2.25: Azure publishing option

Azure offers deployment of Azure Function apps for Windows and Linux environments. Furthermore, it is possible to deploy the app using Docker container. As the app previously created on Azure in this chapter is based on Linux operating system, this underlying option needs to be chosen, as shown in [Figure 2.26](#):

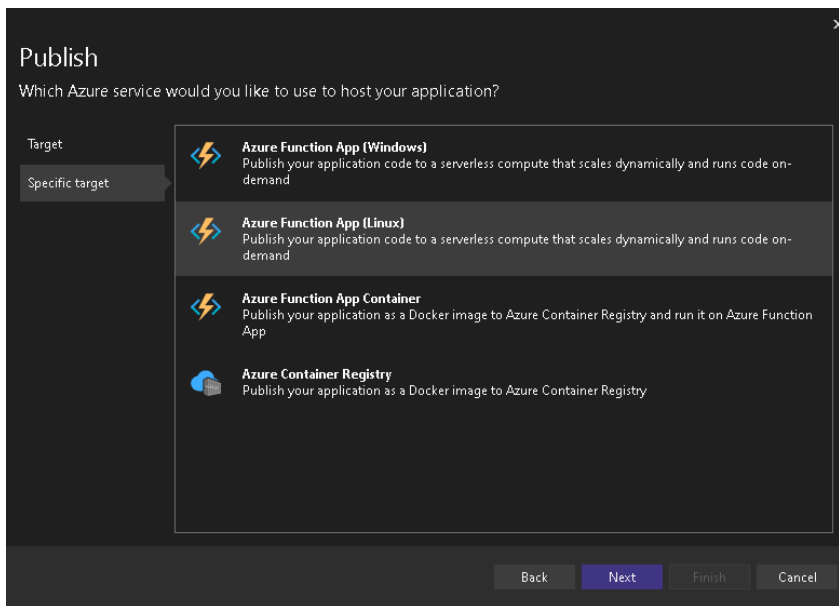


Figure 2.26: Azure function deployment in Linux

Considering the deployment needs to happen in an existing Azure subscription under a Microsoft account, it requires authentication using your own Microsoft credentials. After authenticating into Azure, all the existing Function Apps based on Linux are displayed, as shown in [Figure 2.27](#):

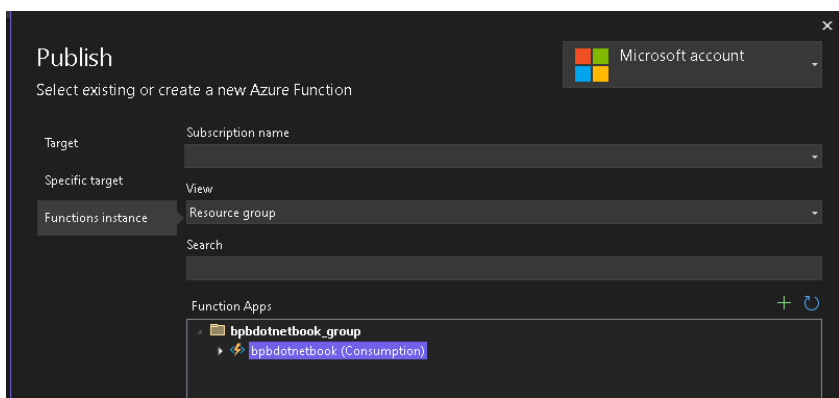


Figure 2.27: Azure Function Apps based on Linux

After finishing the configuration for the publication process, the underlying profile is created, which can be reused for automation

by exporting the related file. The publication should for your function app should look like *Figure 2.28*:

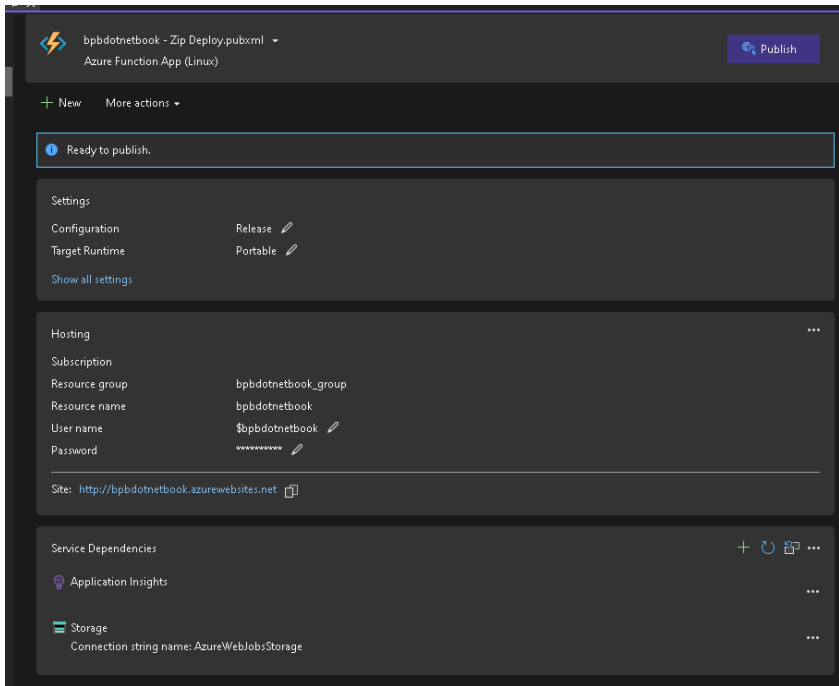


Figure 2.28: Publication profile

After pressing **Publish**, the package for your Azure Function app will be pushed to the infrastructure created on Azure, and you should be able to execute your function using the Azure subdomain for your app. In the context of the one created in this chapter, the URL would be <https://bpbdotnetbook.azurewebsites.net>.

Azure contains a vast range of services that can be consumed via standard REST APIs; including resources for Internet of Things, Artificial Intelligence, Machine Learning and much more. For the majority of services there are mature libraries provided by Microsoft which serves as middleware for C# and other languages. For further information on the Azure services available and the underlying SKDs, please consult the official Azure documentation.

New features in .NET 6

After the release of .NET Core 1.0, which represented the very first .NET version that supported multi-platform development, the .NET platform evolved until .NET 6 to have a single ecosystem that consolidates a mature unification of libraries and SDKs for mobile, web, desktop, cloud and Internet of Things development, having a significant positive impact in terms of performance improvements and multi-platform support. In the next sections you will have the opportunity to walk through the most relevant new features introduced in .NET 6.

Hot reload

Hot reload is the feature built in Visual Studio and .NET platform that allows you to make changes in the codebase and the modifications show up in the application automatically if it is being in execution. This feature is available with Visual Studio 2022 or using the dotnet watch command-line tool in Visual Studio Code. When running an application in debug mode using Visual Studio, you can use the **Hot Reload** button, as highlighted in [Figure 2.29](#), to reflect the changes in the the app automatically without having to stop and application and start that again:

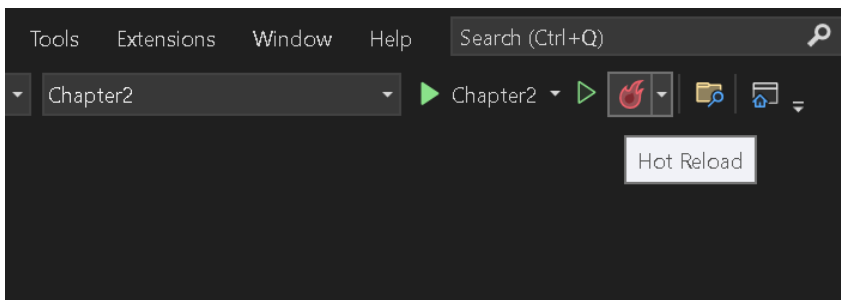


Figure 2.29: Hot Reload

This option increases the productivity significantly in a daily basis, having support to various types of projects in .NET, including Blazor, Web API, MVC, MAUI and others.

.NET MAUI

Multi-platform support is a game-changer for the .NET platform and the .NET MAUI project is on the top of that as it represented a huge effort from Microsoft to allow developers to create a single

application that would be compatible and deployable for Web, Desktop and Mobile platforms, using a single source code. For .NET 6, this project was released in preview.

To start creating MAUI Apps in Visual Studio, you have to install the following workloads:

- Mobile development with .NET
- .NET desktop development
- Desktop development with C + +
- Universal Windows Platform development

If you have already the Visual Studio 2022 installed, you need to use the Visual Studio Installer and check the underlying workloads, as seen in [Figure 2.30](#):

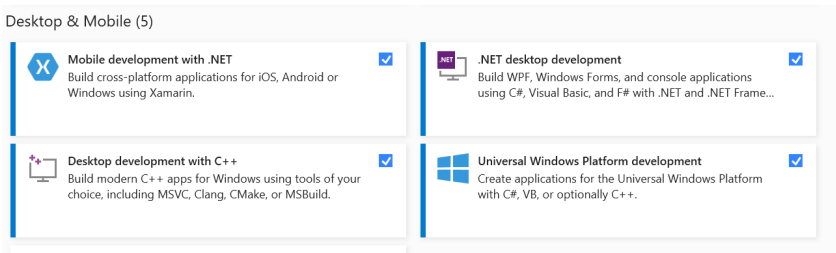


Figure 2.30: Workloads for MAUI project

.NET MAUI represents an a new generation for Xamarin forms for mobile development, but it is much more than that. This new project type is the sign of the direction to where multi-platform is going in terms of modernisation of the .NET platform in general.

HTTP/3

The .NET platform supports HTTP/3 since .NET 6 in preview. This new protocol is related to relevant performance improvements as it uses directly the QUIC technology, which makes connections quicker than previous protocols and allows client devices to switch automatically between Wi-fi and mobile data networks. You can use the HTTP/3 protocol in .NET projects modifying the underlying configuration for the HttpClient objects, as shown in [Figure 2.31](#):


```
var client = new HttpClient()
{
    DefaultRequestVersion = HttpVersion.Version30,
    DefaultVersionPolicy = HttpVersionPolicy.RequestVersionExact
};
```

Figure 2.31: HTTP client configuration for HTTP/3

HTTP3 has a fast response time if compared to HTTP/2, giving to the applications more reliability and a better user experience as this new modern protocol handles more efficiently pack loss and establish connections between the client and server without unnecessary sub-requests involved.

Custom platform checks

.NET 6 introduced new ways of identifying platform and operating systems in case there is a need to write custom code for each of them or change the application behaviour. Furthermore, it is possible to use annotations for methods and properties within a class, in order to determine if the method is supported by a specific platform or not. The platform check method is represented in [Figure 2.32](#):

```
//Chapter 2.3.4 Custom platform checks

if (OperatingSystem.IsLinux())
{
    //Custom code for Linux
}
```

Figure 2.32: Platform check

There is vast number of methods to check the version of operating system and other platform aspects. This is specially useful for multi-platform development, which includes not only the possibility of deploying to different platforms, but to change how the application is going to be have as well in terms of design and functionality.

Conclusion

The .NET platform offers a relevant number of project types to support a wide range of possibilities in terms of innovation, multi-platform development and creation of scalable applications. The history of .NET shows the the platform is still evolving overtime to meet the market needs.

In the next chapter, you will learn basic concepts of **Object-Oriented Programming (OOP)** paradigm, which will give you enough expertise to follow good practices of encapsulation, inheritance, interfaces and SOLID principles, essential concepts for software development.

Points to remember

- .NET 5 represents the unification of the effort behind multi-platform support for the .NET platform
- .NET MAUI is the new generation of multi-platform development in the .NET platform, representing an evolution of Xamarin and consolidation of the support for desktop, mobile and web applications using the same source code.

Multiple choice questions

1. **Which company is responsible for the development of the .NET platform?**
 - A. Apple
 - B. Oracle
 - C. Microsoft
 - D. Google
 - E. Microsoft
2. **In which year was the .NET framework first released?**
 - A. 1998
 - B. 2000
 - C. 2002
 - D. 2005

3. Which of the following is NOT a language supported by the .NET platform for application development?
- A. C#
 - B. F#
 - C. J#
 - D. Ruby

Answers

	C	
	B	
	D	

Questions

1. How has the introduction of .NET Core expanded the capabilities and flexibility of the .NET platform, especially in terms of cross-platform development?
2. In what ways does the .NET framework support and enhance security for application development, especially in web contexts?
3. How does the integration of various programming languages in the .NET platform (like C#, VB.NET, and F#) benefit developers and organizations in terms of versatility and project adaptability?
4. How have recent updates to the .NET platform improved performance and scalability, especially in enterprise-level applications?

Key terms

- **Common Language Runtime (CLR):** The CLR is the virtual machine component of .NET that manages the execution of .NET programs. It provides services like memory management, garbage collection, and security.
- **C# (C Sharp):** C# is a modern, object-oriented programming

language developed by Microsoft as part of the .NET initiative. It's one of the primary languages developers use to create applications on the .NET platform.

- **ASP.NET:** A popular framework for building web applications and web services on the .NET platform. It provides tools and libraries that make it easier to create dynamic websites, web applications, and web services.
- **.NET Core:** A cross-platform (works on Windows, Linux, macOS) version of .NET, which includes the core features of the framework and supports cloud and web app development. It's a modular, high-performance implementation of .NET for creating web applications and services, microservices, and other modern server-side apps.
- **Entity Framework:** An Object-Relational Mapping (ORM) framework for .NET. It allows developers to work with databases using .NET objects without having to focus on the underlying database tables and columns where the data is stored.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Basic Concepts of Object-Oriented Programming in C#

Introduction

Software development is an abstract process that correlates with the real world throughout the implementation of business and system requirements presented in the software in a genuinely corresponding way, generally using the object-oriented programming paradigm.

The concept behind this paradigm helped the software industry for decades to create stable, readable, and extensible code for enterprise applications, making the coding process more accessible for developers and bringing an intelligent design to be used by multiple programming languages.

Furthermore, the discussion around **Object-Oriented Programming (OOP)** brought to the market many additions and contributions to the software engineering field, such as SOLID principles concept and design patterns, which increased the overall

quality of software development worldwide and gave the possibility of building robust and scalable applications that are easy to maintain, bringing the software engineering to a next level in terms of techniques and professionalism.

Because of all these facts, any developer needs to learn this paradigm properly to apply its concepts in real and complex scenarios, a mandatory requirement for almost all software developer roles in the market.

With this chapter, you will have the opportunity to learn the essential concepts of object-oriented programming, such as classes, constructors, methods, abstractions, inheritance, and interfaces. Additionally, this chapter will introduce explanations and examples of design patterns and SOLID principles.

Structure

In this chapter, we will discuss the following topics:

- Classes, constructors, and methods
- Static classes and methods
- Structs
- Abstract classes, inheritance, and interfaces

Objectives

After studying this unit, you should be able to understand object-oriented programming concepts and discuss essential aspects of design patterns. By the end of this chapter, you should be able to create basic applications using object-oriented programming and use SOLID principles in real projects.

Classes and access modifiers

Starting from scratch with **Object-Oriented Programming (OOP)** concepts, you must know that any software in the market and any code implementation using a modern programming language have a specific goal to be aimed at, an explicit intention of solving a real problem surrounded by business needs and non-functional requirements. There are many ways of solving a single problem in

software engineering, and keeping and writing understandable code is vital to the success of any project, maintaining correlation with the code implementation and the real-world concepts that the software tries to address. The OOP helps us keep this correlation between business requirements and logical implementation, facilitating the comprehension of the code so that it can be interpreted not only by machines but by developers as well.

The C# language has fully supported OOP since its first version, and every application that users of .NET must apply this paradigm as all the native libraries of the platform are strictly based on this concept, following this pattern. Furthermore, the .NET platform benefits from the advanced concepts of design patterns and SOLID principles, which will be explained further in this book using examples based on real scenarios.

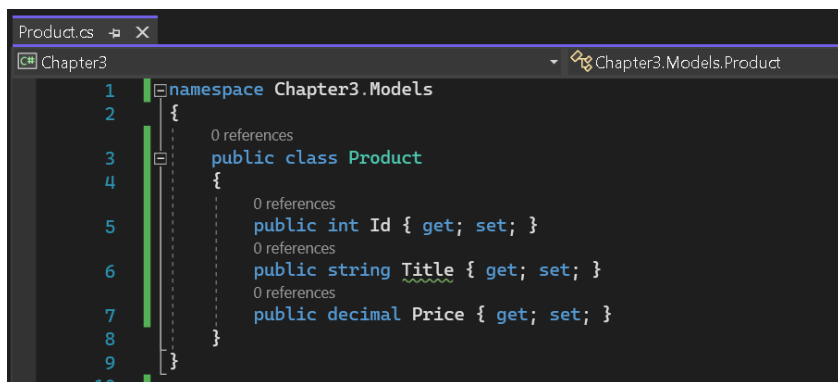
The OOP allows us to organize the implementation of software requirements around objects, representing the behavior and state of something that needs to be an abstraction in the system. An object can contain properties and methods that are responsible for keeping and manipulating the state of objects and allows the execution of operations related to the objects themselves. In C# and other languages based on the OOP paradigm, an object is defined by a class that should have similar characteristics to the real-world entities stated in the business requirements, following a faithful representation.

For better understanding, imagine a scenario where an online store needs to be developed using C# language and .NET. In this fictional study case, the system would implement the following requirements:

- Allow users to register products and product types.
- Every product should have a title and a unique number to identify the product.
- Users are considered any person who access the online shop but do not necessarily complete a purchase.
- Once a user makes a purchase in the online shop, the same user should be registered as a customer with additional information for billing purposes.
- The products can contain different properties based on their

type.

Considering the given scenario, if we would start planning the design of the necessary code to keep everything well organized, we must create the classes first, with the most fundamental properties common to any concrete object that will be created using these classes. Starting from the model for the product, the initial implementation of the class would have the implementation as seen in [Figure 3.1](#):

The image is a screenshot of a code editor window titled 'Product.cs'. The editor shows a C# code snippet for a class named 'Product' within the 'Chapter3.Models' namespace. The code is as follows:

```
1 namespace Chapter3.Models
2 {
3     public class Product
4     {
5         public int Id { get; set; }
6         public string Title { get; set; }
7         public decimal Price { get; set; }
8     }
9 }
```

The code is displayed on a dark background with syntax highlighting. Line numbers 1 through 10 are visible on the left side of the editor. The 'Product' class is defined with three public properties: 'Id' of type 'int', 'Title' of type 'string', and 'Price' of type 'decimal'. Each property has a getter and a setter. The namespace is 'Chapter3.Models'.

Figure 3.1: Product class

In this case, the fields **Id**, **Title**, and **Price** represent the properties of the **Product** class, and each has its type, such as **integer**, **string**, and **decimal**. You must note that there is a keyword before each property type. The keyword means the access modifier or scope of the property for the entire solution. The OOP contains the following access modifiers:

- **Public**: The class or property can be freely accessed by other classes, even by the ones that are present in the same assembly.
- **Private**: The property can be accessed only by the same class where the property is located.
- **Protected**: The property can be accessed only within the same class or any derived classes.
- **Internal**: The property or class can be accessed by other classes, but with the restriction for the classes to be in the same assembly.

- Protected internal: The class or property can be accessed by any other class in the same assembly and by all derived classes, even if they are not in the same assembly.

Depending on the access modifier used, the access level changes, as can be seen in [Figure 3.2](#):

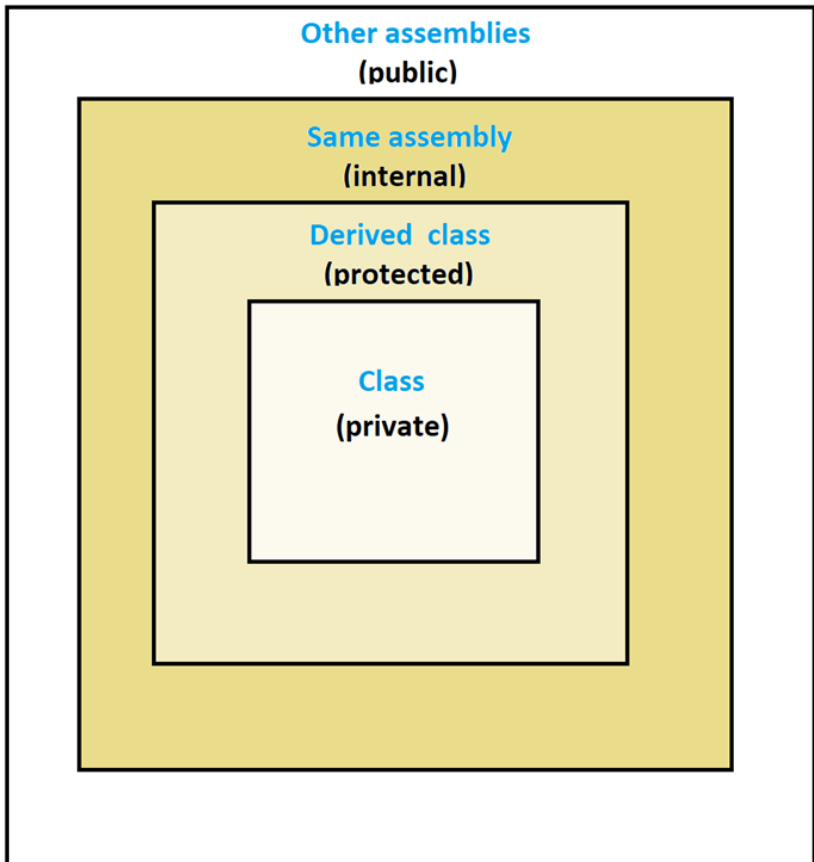


Figure 3.2: Access modifier

OOP has principles that enforce good coding practices for modern programming languages, with the additional purpose of not violating the mechanisms that should be followed generally in any software based on the OOP paradigm, such as encapsulation, inheritance, polymorphism, and reusability. The following sections of this chapter contain detailed information on each of these

practices.

Encapsulation

Encapsulation within the OOP paradigm avoids external interference on the state of objects, protecting the access to properties of a class and its methods. To meet this objective successfully, understanding the meaning of the access level for the properties of the classes was mentioned previously in this chapter. The explicit control of access level for properties, methods, and classes ensures that other developers working on the project cannot violate the desired boundaries for classes, including aspects on how each class manages the state for the underlying objects. When there is a low dependency between classes, it increases aspects of testability and maintainability for projects as a whole.

Inheritance

The concept of inheritance in the OOP paradigm is strongly related to the principle of reusability in software development. It allows us to share properties and methods among classes that have similar purposes. A class that inherits from another class is called a **derived class**, and the base class is usually called a parent class in this context.

The C3 language allows us to specify only one inheritance associated with a class signature, but it is possible to use interfaces to simulate multiple inheritance implementation when needed. In terms of a practical example of inheritance, we can extend the scenario already given in this chapter for the Online Store, which could contain many types of different products that would share common properties and methods between them, with the possibility of having properties that would apply only to specialized classes.

In this context, the **Movie** class is a type of product and shares similar characteristics to other products, such as **Id**, **Title**, and **Price**. Still, it also has specific properties that are not applicable to other classes. Considering the **Product** class demonstrated in Figure 3.1 in this chapter that contains already the common base properties, the Movie class can inherit from the **Product** class, as seen in [Figure 3.3](#):

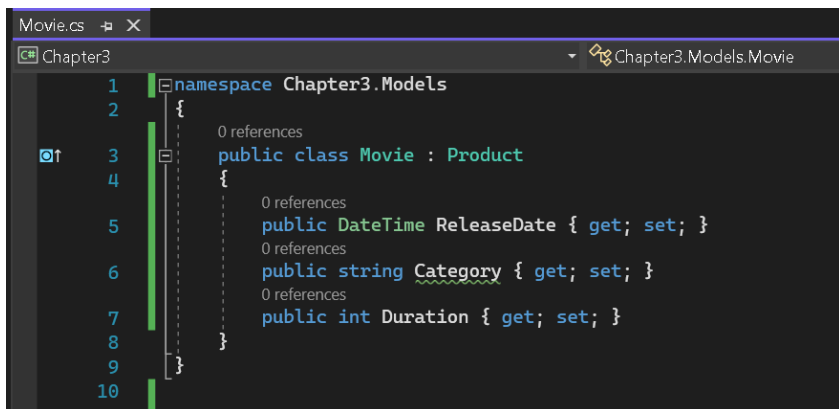


Figure 3.3: Access modifier

In the signature of this **Movie** class, inheritance is declared from the **Product** class, which means that the **Movie** class will contain all the properties that exist in the **Product** class among other specific fields from its class, such as **Release Date**, **Category**, and **Duration**. The decision to inherit or not from another class depends on the context of the project and business requirements. If different classes have a clear correlation, it might indicate that inheritance could be suitable. This OOP feature helps developers save time writing code across multiple classes for an application, and it also helps to keep the consistency of the coding design decided for the project.

Reusability

Generally, the costs of developing any software are measured by the number of hours spent writing code, among other activities such as business analysis, software design, and project management. Considering the time spent in coding represents one of the essential points of the project to succeed, it is crucial to use techniques that allow us to save time in building software and optimizing the coding process. Reusability is one of the essential concepts and benefits of the OOP paradigm because a correct design and architecture directly correlate to the costs of software projects.

Going back to the Online Store scenario demonstrated in this chapter, as the sample already contains a base **Product** class, if the Online Store had a vast number of products of different types, the

intelligent use of the base class for all products would save a significant time in the implementation of the entire project. The same concept of reusability and its benefits apply to more complex scenarios where not only simple properties and methods can be reused, but whole libraries and packages can be shared across multiple projects to maximize the reuse of classes, components, and features.

Polymorphism

Polymorphism allows us to overwrite a method from a parent class, sharing a standard interface for multiple classes to give integrity and consistency in the software architecture. As its name suggests, polymorphism is the possibility of assuming many forms for a common method. In the Online Store example, imagine that each product could have its way of applying discounts based on custom rules. In this case, the **Product** base class would have a standard method to calculate the discount, but all the derived classes would have the possibility of overwriting the behavior of the discount method according to its convenience.

For instance, the **Product** parent class could have a method to apply a seven percent discount by default, as seen in [Figure 3.4](#):

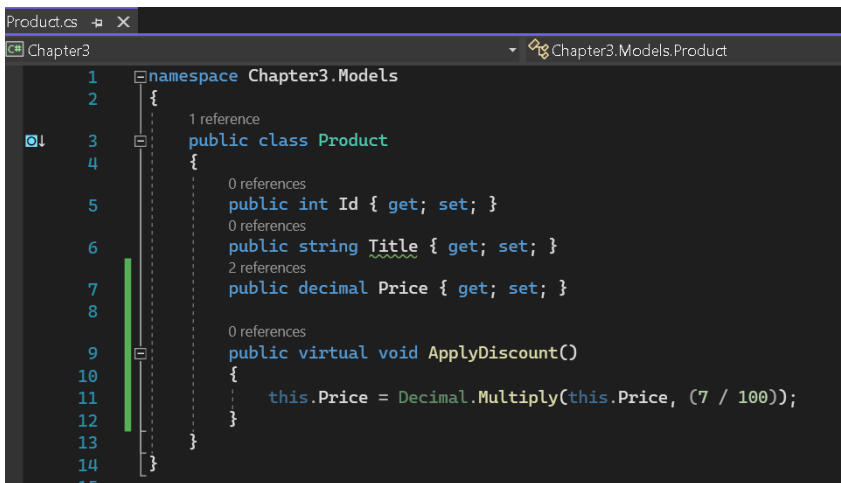


Figure 3.4: Method to apply the discount on the Product class

On the other hand, the **Movie** class, which is a derived class from

Product, can overwrite the implementation of the **ApplyDiscount** method, as seen in *Figure 3.5*:

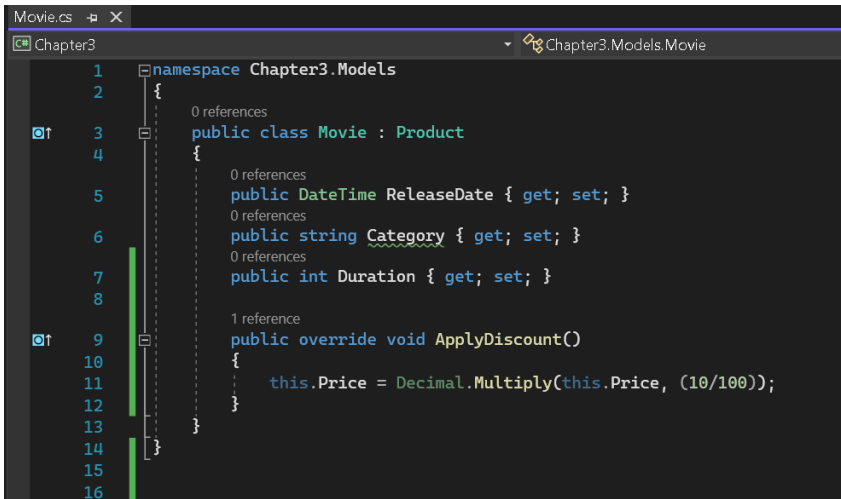


Figure 3.5: Method to apply the discount on the Movie class

The **Movie** class applies a different policy for the discount by applying a value of ten percent, changing the behavior of the same method inherited from the **Product** class. However, it is possible to change the implementation of a method from the parent class only if the method is marked with the virtual modifier in the parent class, as highlighted in *Figure 3.6*:

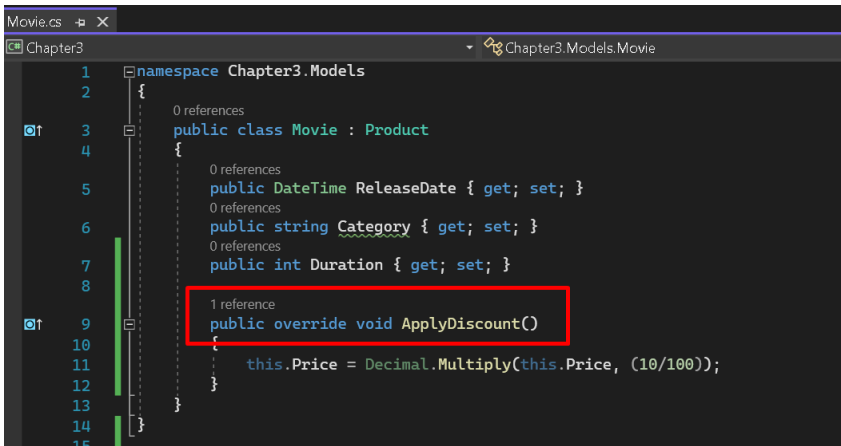


Figure 3.6: Method to apply the discount on the Movie class

The **override** keyword needs to be used in this case to indicate that the underlying from the parent class will not necessarily be used in the same way.

Partial class

To keep good practices of coding and a well-structured project, usually, the entire implementation of the class is presented in the same file, following a convention that the filename is the same as the class name, as highlighted in [Figure 3.7](#):

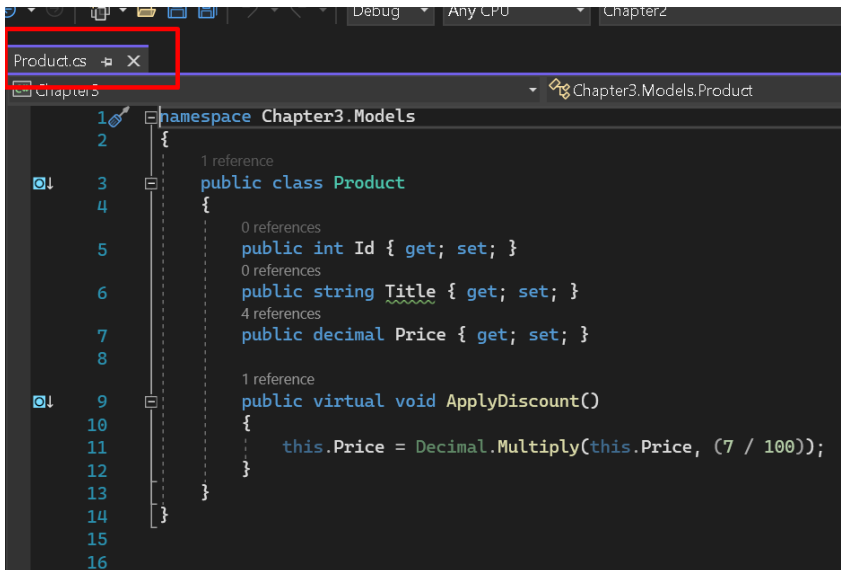
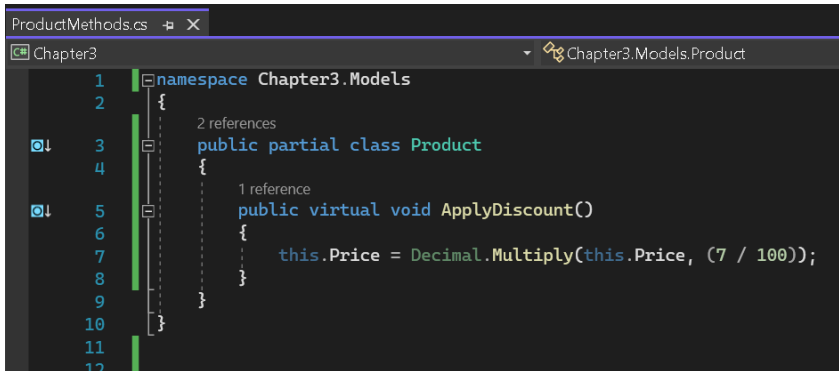


Figure 3.7: Filename pattern for classes

There are cases where a class contains many code lines and complex maintenance when multiple developers need to work simultaneously on the same file. In this scenario, it is convenient to split the class into multiple files to avoid merging conflicts and to have a better logical separation for understanding purposes.

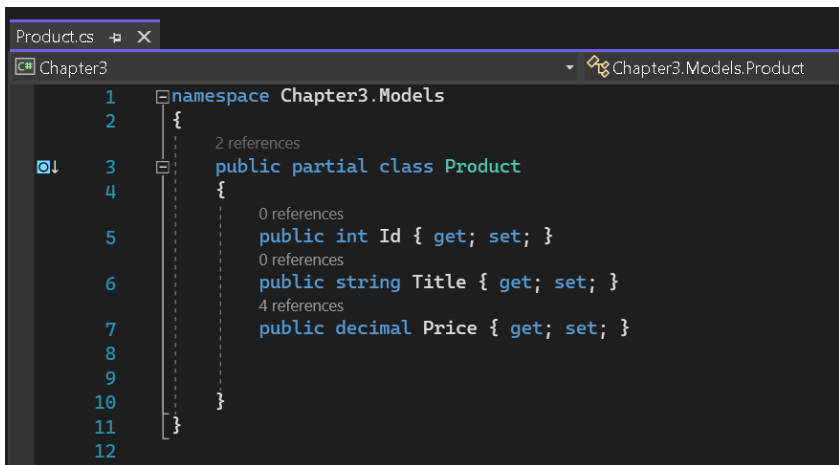
Although the class is presented in more than one file, the compiler combines them as a single class. In the following code sample, there are two files regarding the **Product** class, one containing the fields and another that has the methods for the class, as seen in [Figure 3.8](#):



```
1 namespace Chapter3.Models
2 {
3     public partial class Product
4     {
5         public virtual void ApplyDiscount()
6         {
7             this.Price = Decimal.Multiply(this.Price, (7 / 100));
8         }
9     }
10 }
11
12
```

Figure 3.8: Partial product class

The **partial** keyword is used in the class signature, and the same process applies to all the other partial classes for the **Product** class. As the given partial **Product** class has only the method responsible for applying the discount and the rest of the implementation for the class was placed in another file, as seen in [Figure 3.9](#):



```
1 namespace Chapter3.Models
2 {
3     public partial class Product
4     {
5         public int Id { get; set; }
6         public string Title { get; set; }
7         public decimal Price { get; set; }
8     }
9 }
10
11
12
```

Figure 3.9: Partial Product class with properties

The use of partial classes is quite flexible, but it is recommended to use this only in cases where the class has a reasonable big size to justify splitting them into partial classes. Furthermore, suppose a class contains a lot of code lines. In that case, usually, it is a clear indication that the class should be refactored to follow the **Single Responsibility Principle (SRP)**, a good practice stated by the

SOLID principles that will be explained further in this book.

Constructor

In OOP, the constructor is the method called by the compiler when an instance of an object is created. A constructor has the same name as the class and does not have any associated return. In the Online Store scenario, considering there is a class called **Product**, every time this class needs to be used, a new object instance is created with the default constructor for the class being called by default, as seen in [Figure 3.10](#):

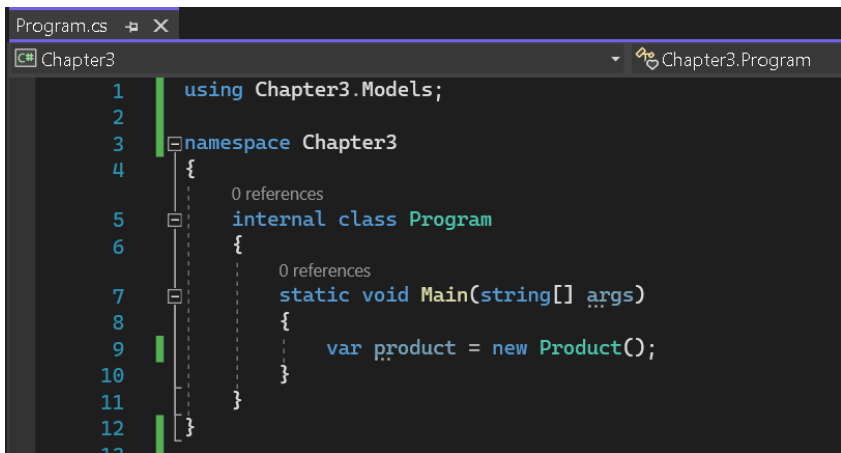


Figure 3.10: Product object instance

When the **new** keyword is used, it indicates to the compiler that a new instance of the **Product** class should be materialized as an object, which will be placed in a new space allocated in the memory. Additionally, the new object is initialized in this operation, and the constructor method is mandatorily executed. You can specify multiple constructors for a class if they have different arguments, with all of them having the same method name, as demonstrated in [Figure 3.11](#):

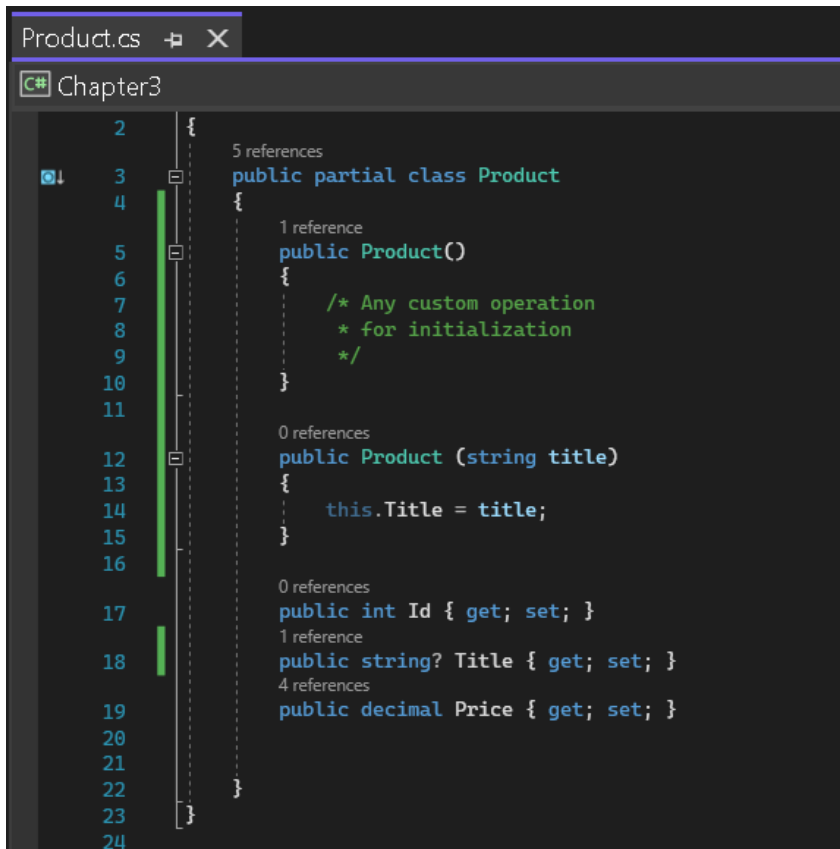


Figure 3.11: Multiple constructors

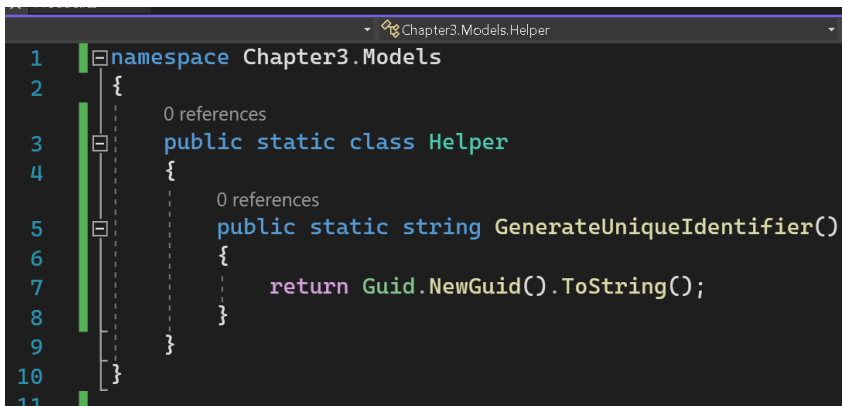
Usually, constructor methods are used to set the initial state of objects. If the constructor method is not specified, the compiler runs the default constructor method, which is the empty version without any argument. It is essential to understand precisely how constructors work because the state management of objects is one of the main aspects to consider when applying the OOP paradigm following good practices.

As demonstrated in [Figure 3.11](#), it is possible to pass parameters in the constructor, which is usually done when it is necessary to make operations when an object is initialized, such as setting the initial state for class properties or injecting objects from other classes, using the concept of dependency injection.

Static classes

In C# language, a static class represents a class that cannot have any object created. This means the system keeps in the memory a single instance of the class that is used for the entire system.

Considering that characteristic, it is impossible to use the keyword `new` when there is the intention to use the underlying class. Usually, static classes are used when they do not need a dynamic state. For instance, the Online Store scenario could have a static class to make operations that are common for the entire application and are not related to the existing main entities or classes. It can be applied to methods regarding type conversions, math operations, and others. The following code sample demonstrated in [Figure 3.12](#) contains an implementation of a static class called `Helper`, which has a method to return a unique identifier:

The image is a screenshot of a code editor window titled 'Chapter3.Models.Helper'. It shows the following C# code:

```
1 namespace Chapter3.Models
2 {
3     0 references
4     public static class Helper
5     {
6         0 references
7         public static string GenerateUniqueIdentifier()
8         {
9             return Guid.NewGuid().ToString();
10        }
11    }
```

Figure 3.12: Static Helper class

As shown in [Figure 3.12](#), a `static` keyword transforms the `Helper` class into a static class. Considering it is not possible to create an instance of a static class, the compiler throws an error if an implementation tries to violate that principle, as seen in [Figure 3.13](#):

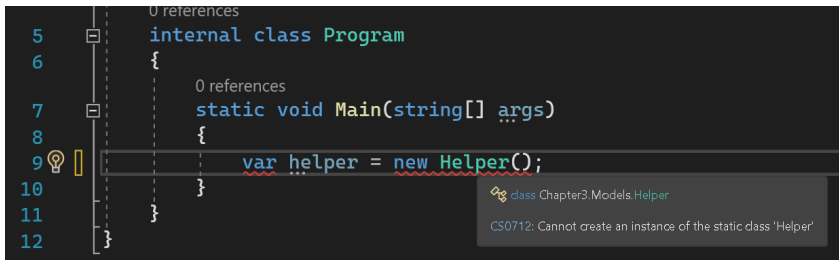


Figure 3.13: Compiler error for static classes

To use the methods present in the static class, you call them directly without using the **new** keyword, as demonstrated in [Figure 3.14](#):

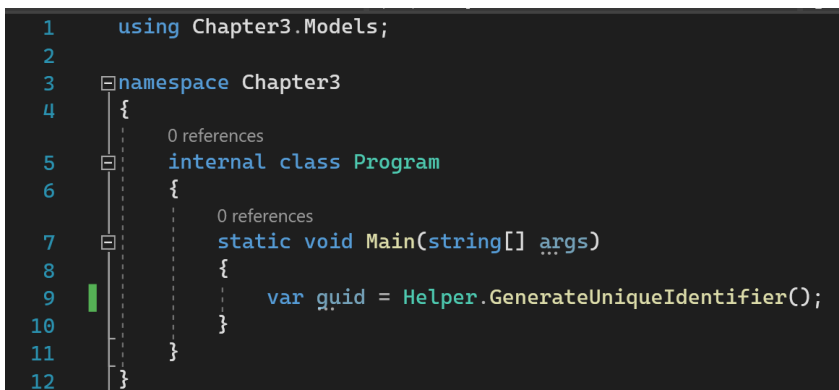


Figure 3.14: Correct use of static classes

Usually, when a static class is defined, all its members follow the same signature in terms of being marked as static members.

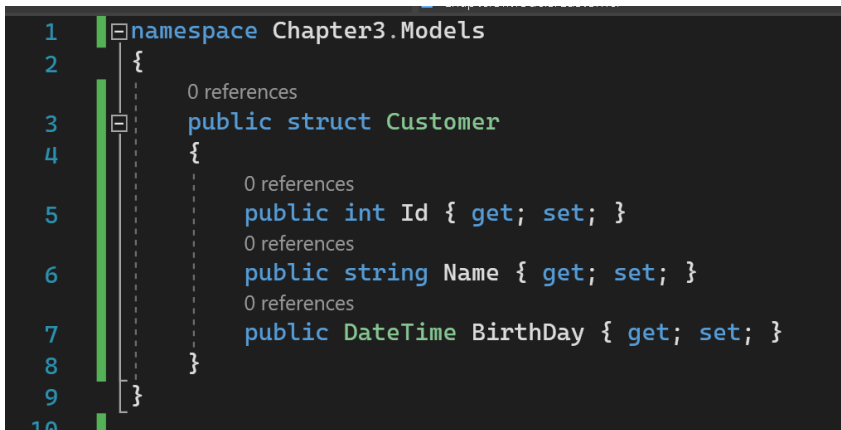
Structs

Structs in C# have a similar implementation to classes, including properties and methods. However, the main difference resides in how they are kept in the memory when the C# program is executed. A conventional class is considered a reference type in C#, which means that this type does not store its value in the exact memory location as its reference in the memory. For classes, if the same object instance is used multiple times in a program, the memory finds the reference in the memory for the underlying object when it needs to be accessed and not the state of an object directly.

On the other hand, structs contain the reference and the value stored all together, making access to the values significantly faster than reference types in general. Furthermore, structs have restrictions if compared to classes, as follows:

- It is not possible to use structs for inheritance
- It is not allowed to use an empty constructor for a struct. It must have a constructor with parameters.
- When an instance is created, it is required to set values to local variables.
- The instance of the object created from a **struct** is removed from the memory once the object's method completes its process.

The syntax of a structure is quite similar to classes, as seen in [Figure 3.15](#):

A screenshot of a code editor showing C# code. The code is enclosed in a namespace `Chapter3.Models`. Inside this namespace, there is a `public struct Customer` definition. The struct has three public properties: `int Id`, `string Name`, and `DateTime BirthDay`, each with `get` and `set` accessors. The code is formatted with syntax highlighting: namespaces and keywords in blue, types in green, and property names in white. Line numbers 1 through 10 are visible on the left side of the editor.

```
1 namespace Chapter3.Models
2 {
3     0 references
4     public struct Customer
5     {
6         0 references
7         public int Id { get; set; }
8         0 references
9         public string Name { get; set; }
10        0 references
        public DateTime BirthDay { get; set; }
    }
}
```

Figure 3.15: Struct example

A struct must have a constructor with parameters, as demonstrated in [Figure 3.16](#):

```

1 namespace Chapter3.Models
2 {
3     1 reference
4     public struct Customer
5     {
6         0 references
7         public Customer(int id, string name, DateTime birthday)
8         {
9             Id = id;
10            Name = name;
11            BirthDay = birthday;
12        }
13        1 reference
14        public int Id { get; set; }
15        1 reference
16        public string Name { get; set; }
17        1 reference
18        public DateTime BirthDay { get; set; }
19    }
20 }

```

Figure 3.16: Struct constructor

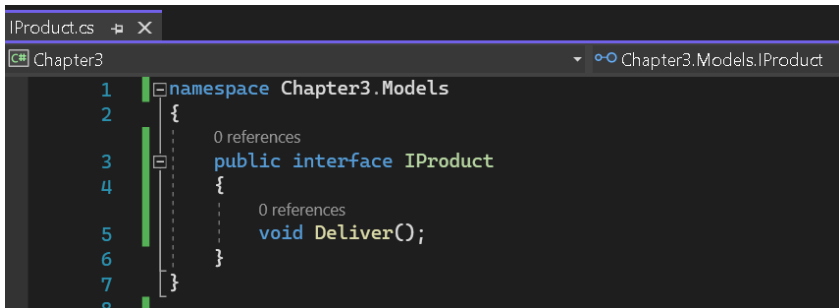
The use of structs is quite helpful when a specific part of the application requires high performance in terms of memory management, being an excellent alternative for models that have a pre-defined state and the life-cycle in the program execution is short enough.

Interfaces

In OOP, an interface represents a contract that should be followed for all classes with reference to the interface on their signature. In other words, an interface contains everything mandatory to exist in a class, including properties and methods. This feature is significant in correctly implementing business requirements and enforcing consistency in the development process.

In C# language, the user can create an interface with the corresponding keyword before the interface's name. By nature, interfaces do not contain implementation, which is the responsibility of the classes that implement those. C# 8.0 introduced the possibility of specifying default interface methods, which would contain implementation, but this seems to be an anti-pattern as the primary purpose of interfaces is to establish a contract for classes, delegating to them the behavior for the methods.

Referring to the Online Store example, considering there would be a requirement that all product types should implement a method referent to product delivery, an interface could be specified to ensure that all the classes related to product type will implement the underlying delivery method. A basic interface for the product model would have a similar representation seen in [Figure 3.17](#):



The screenshot shows a code editor window titled 'IProduct.cs'. The left sidebar shows a project structure with 'Chapter3' expanded. The main editor area displays the following C# code:

```
1 namespace Chapter3.Models
2 {
3     public interface IProduct
4     {
5         void Deliver();
6     }
7 }
8
```

The code defines a namespace 'Chapter3.Models' containing a public interface 'IProduct' with a single method 'Deliver()'.

Figure 3.17: Product interface

In this **Product** interface, there is a specification that enforces the implementation of the **Deliver** method by all the classes with reference to the interface. This reference can be done after the class name, as seen in [Figure 3.18](#):

```

1  namespace Chapter3.Models
2  {
3      4 references
4      public partial class Product: IProduct
5      {
6          0 references
7          public Product()
8          {
9              /* Any custom operation
10             * for initialization
11             */
12         }
13
14         0 references
15         public Product (string title)
16         {
17             this.Title = title;
18         }
19
20         0 references
21         public int Id { get; set; }
22         1 reference
23         public string? Title { get; set; }
24         4 references
25         public decimal Price { get; set; }
26     }

```

Figure 3.18: Product interface reference

Once the interface is being used, if the implementation of any method from the interface is missing, the compiler accuses that, as shown in [Figure 3.19](#):

```

4 references
public partial class Product: IProduct
{
    0 references
    public Product()
    {
        /* Any custom operation
        * for initialization
        */
    }
}

```

interface Chapter3.Models.IProduct

CS0535: 'Product' does not implement interface member 'IProduct.Deliver()'

Show potential fixes (Alt+Enter or Ctrl+.)

Figure 3.19: Compiler error for missing implementation

C# 10 introduced the concept of static abstract members for interfaces, allowing the implementation of custom versions of the

property by types that implement the interface. For instance, you can have a member called `IsDeliverable` in the `Product` interface marked as static and abstract. In this case, it is up to the classes to define the value for the static property, as seen in [Figure 3.20](#):

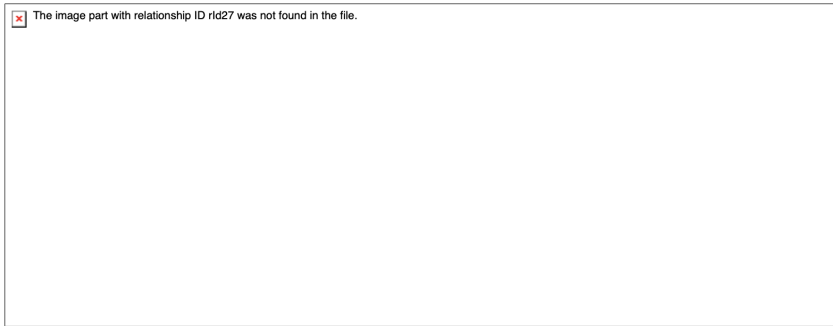


Figure 3.20: Static abstract interface

In this case, it is up to the classes to determine what would be the value for the static member, as seen in [Figure 3.21](#):

```
0 references
3 public class Movie : IProduct
4 {
5     1 reference
6     public static bool IsDeliverable => false;
7
8     0 references
9     public DateTime ReleaseDate { get; set; }
10    0 references
11    public string Category { get; set; }
12    0 references
13    public int Duration { get; set; }
14
15    1 reference
16    public void Deliver()
17    {
18        //Any custom implementation for the Deliver method
19    }
20 }
```

Figure 3.21: Static member implementation

The `Movie` class implemented the static member enforced by the `Product` interface, but the value is set to `false`, which may not be the case for other classes that would implement the same interface.

The use of interfaces can reduce complexity and provide a higher level of maintainability in the code once they give consistency to classes and allow the application to apply a pattern across the application. Additionally, the interfaces are essential to make unit and integration tests easier because the dependency between classes can be mocked if the correct concept of dependency injection is used.

Conclusion

As seen in this chapter, the OOP paradigm is the base of the C# language. It is one of the most essential aspects of modern software development, representing something every developer must know and be proficient in. The correct use of this paradigm allows companies to create testable, scalable, and stable projects and helps to reduce the cost of the development process by minimizing the risks of the introduction of issues and bugs. All the logical implementations for business and system requirements can be organized in classes, which can be reused, tested, and extended over time.

This chapter teaches you how to create classes, properties, methods, and definitions of access levels. Furthermore, you could familiarize yourself with interfaces, structs, and new features introduced in recent versions of the C# language. Finally, you learned important concepts of OOP, such as encapsulation, polymorphism, and inheritance.

The next chapter explains in details all the SOLID principles using the latest version of C# language.

Points to remember

- Objects represent an instance of a class in OOP.
- Interfaces are contracts that make the implementation of methods and properties within classes mandatory.
- Inheritance is the best way of reusing code in the OOP paradigm.

Multiple choice questions

1. Which operation is not allowed in C# for OOP?
 - A. Interfaces
 - B. Multiple inheritances
 - C. Static classes
 - D. Access modifiers
2. How many interfaces can be used associated with a single class?
 - A. One
 - B. Two
 - C. Five
 - D. Unlimited
3. What is the access level that restricts access to the assembly of the same assembly?
 - A. Protected
 - B. Public
 - C. Internal
 - D. Private

Answer

	B	
	D	
	C	

Questions

1. Explain the difference between classes and structs.
2. Explain the meaning of SOLID principles.

Key terms

- **Class:** A blueprint or template for creating objects (specific instances). In C#, classes define properties, methods, events,

and more that act as the structure for the objects.

- **Object:** An instance of a class. It's a real-world representation of a class blueprint.
- **Encapsulation:** The bundling of data (attributes) and methods (functions) that operate on the data into a single unit, ensuring that the object's internal state is hidden from the outside.
- **Inheritance:** A mechanism where a new class derives properties and behaviors (methods) from an existing class. It promotes the reuse of code.
- **Polymorphism:** The ability of a single function or method to work in different ways based on the object it's acting upon. In C#, this is often achieved via method overriding and method overloading.

CHAPTER 4

SOLID Principles in C#

Introduction

With this chapter, we are starting our journey into the best practices of object-oriented programming using the C# language. Any software project could face similar challenges in terms of scalability, maintenance, and regression issues. To facilitate the development process, it is highly recommended to follow the five most famous principles of the OOP paradigm: **Single-Responsibility Principle (SRP)**, **Open-Closed Principle (OCP)**, **Liskov Substitution Principle (LSP)**, **Interface Segregation Principle (ISP)**, and **Dependency Inversion Principle (DIP)**.

You will learn to apply all these standard practices in real projects using the C# language. The knowledge of SOLID principles is mandatory to start with design patterns, the book's primary purpose. Considering the high complexity that a software implementation might have in terms of coding structure, it is essential to understand these principles, allowing you to identify them in existing projects and third-party libraries.

Structure

In this chapter, we will discuss the following topics:

- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Dependency Inversion Principle

Objectives

After studying this unit, you should be able to understand the SOLID principles and identify the SOLID principles in existing projects. You should also be able to apply the SOLID principles in basic projects in .NET 7 and C# 11.

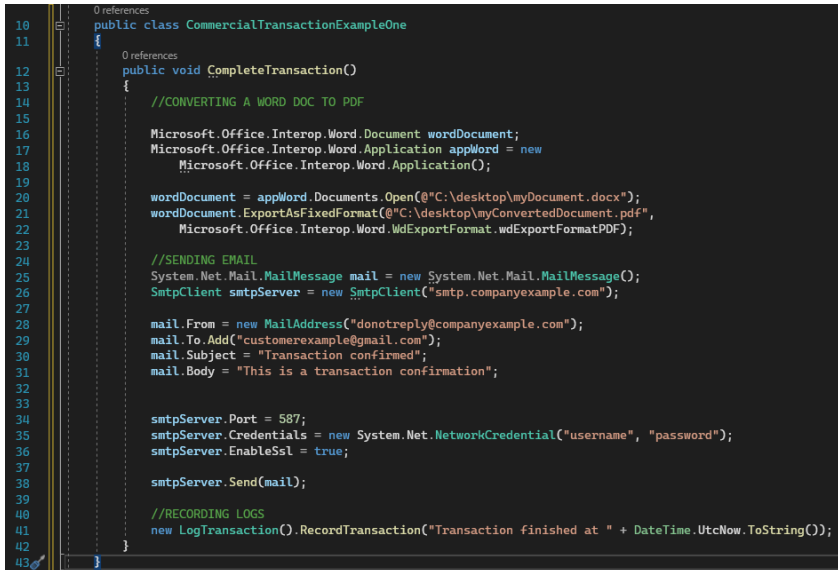
The Single Responsibility Principle

The SOLID principles are good practices that should be followed by all software developers responsible for building high-quality applications using the OOP paradigm. The principles contribute to having well-written code, providing maintainability and extensibility. Software development is not only about visible software functionalities but about having high standards in software architecture and engineering.

It does not take much effort to find software projects that have failed in delivery, quality, and user expectations because of technical problems and limitations regarding the ability to extend the capabilities to extend the software to new functionalities over time. In any software development cycle, the costs demanded changes in the project get higher when changes are made after the software has been released to users. Changes to any existing functionality would likely introduce new issues to the project; at the same time, adding new features becomes more complicated.

For these challenges in software development, keeping the code as clean as possible and following good practices of software development is essential to have a successful project and guarantee that the project is sustainable long-term in terms of support and evolution.

For instance, imagine a scenario in which a commercial system contains a specific routine that converts a document to PDF, sends it by email to users, makes a financial transaction, and records logs in the database regarding all the operations. In this simple use case, four different business requirements must be accomplished in a single routine. Without following any SOLID principle, the class to make these operations would look like the representation in [Figure 4.1](#):



```
0 references
10 public class CommercialTransactionExampleOne
11 {
12     0 references
13     public void CompleteTransaction()
14     {
15         //CONVERTING A WORD DOC TO PDF
16         Microsoft.Office.Interop.Word.Document wordDocument;
17         Microsoft.Office.Interop.Word.Application appWord = new
18             Microsoft.Office.Interop.Word.Application();
19
20         wordDocument = appWord.Documents.Open(@"C:\desktop\myDocument.docx");
21         wordDocument.ExportAsFixedFormat(@"C:\desktop\myConvertedDocument.pdf",
22             Microsoft.Office.Interop.Word.WdExportFormat.wdExportFormatPDF);
23
24         //SENDING EMAIL
25         System.Net.Mail.MailMessage mail = new System.Net.Mail.MailMessage();
26         SmtplibClient smtpServer = new SmtplibClient("smtp.companyexample.com");
27
28         mail.From = new MailAddress("donotreply@companyexample.com");
29         mail.To.Add("customerexample@gmail.com");
30         mail.Subject = "Transaction confirmed";
31         mail.Body = "This is a transaction confirmation";
32
33
34         smtpServer.Port = 587;
35         smtpServer.Credentials = new System.Net.NetworkCredential("username", "password");
36         smtpServer.EnableSsl = true;
37
38         smtpServer.Send(mail);
39
40         //RECORDING LOGS
41         new LogTransaction().RecordTransaction("Transaction finished at " + DateTime.UtcNow.ToString());
42     }
43 }
```

Figure 4.1: Code before applying the Single Responsibility Principle

The given implementation contains three distinct responsibilities:

- Convert a document to PDF format
- Send emails to users
- Record the transaction in a log file

According to the method called **CompleteTransaction** in this implementation, the primary purpose of this routine would be the execution of the commercial transaction. Still, the actual code is doing much more than that: sending emails and converting files to PDF format. This clearly indicates that the SRP is not being followed, increasing the possibility of introducing regression bugs in this code over time, as any change in one of the multiple parts of

the method can easily affect the execution of the entire method. Additionally, combining multiple responsibilities into the same method makes it harder to reuse the method in other software parts.

An alternative would be refactoring this code, hiding the implementation of everything secondary to the method, and keeping only the code associated with transaction completion, as shown in *Figure 4.2*:

The image shows a screenshot of a code editor with a file named 'CommercialTransaction.cs'. The code is written in C# and follows the Single Responsibility Principle. It includes a 'using System;' statement, a 'namespace SOLIDPrinciplesBook' declaration, and a 'public class CommercialTransaction' definition. Inside the class, there is a 'private void CompleteTransaction()' method. This method contains three lines of code: 'new Document().ConvertToPDF(@"C:\desktop\myDocument.docx");', 'new Email().Send();', and 'new LogTransaction().RecordTransaction("Transaction finished at " + DateTime.UtcNow.ToString());'. The code is formatted with line numbers on the left and indentation for the class and method blocks.

Figure 4.2: Single responsibility refactor

This new version of the exact implementation has significant improvements in terms of coding. The complexity related to emails and conversion to PDF was moved to other classes and can be reused in the parts of the software. Furthermore, the method **CompleteTransaction** contains fewer code lines, making it easier to understand their purpose. The most crucial point of the SRP is that a class should have only one reason to change. After the refactor process, the code is split into three different classes, and they can be distinctly changed without affecting the others. Therefore, the primary class is not violating the principle anymore.

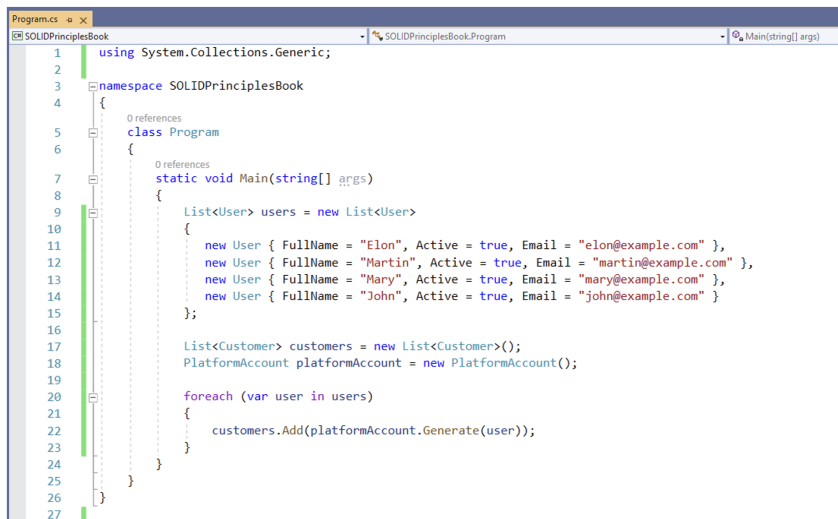
The open-closed principle

The OCP is the second of the SOLID acronym, and the idea behind this principle is that an object should be open to an extension but closed for changes in its behavior. Any software has a huge amount of business requirements converted into code. The requirements can frequently change because of business requests, new regulations, or simply because of a better understanding of the purpose of the

software. However, many requirement changes should not represent or require modifications to the code every time. Suppose a specific routine in the code is being changed several times. In that case, it probably means that the design or pattern used for developing this part of code is not clear enough, and it is not following the best practices of software development.

Furthermore, each change in the software could cause an impact on production environments by adding errors to existing functionality, which should be avoided as much as possible. Following the OCP, if a new requirement is requested and requires changing an existing class or method in the code, the class should be extended to contemplate this new request and not changed. Therefore, adding new requirements should be considered new functionalities and represent extensions instead of modifications.

For instance, imagine the following scenario. There is a system that allows users to create accounts and have access to online movies on a streaming platform. At the first moment, a single account type has the same behavior and requirements for every user. The following [Figure 4.3](#) represents a simple implementation of that:



```
1 using System.Collections.Generic;
2
3 namespace SOLIDPrinciplesBook
4 {
5     0 references
6     class Program
7     {
8         0 references
9         static void Main(string[] args)
10        {
11            List<User> users = new List<User>
12            {
13                new User { FullName = "Elon", Active = true, Email = "elon@example.com" },
14                new User { FullName = "Martin", Active = true, Email = "martin@example.com" },
15                new User { FullName = "Mary", Active = true, Email = "mary@example.com" },
16                new User { FullName = "John", Active = true, Email = "john@example.com" }
17            };
18            List<Customer> customers = new List<Customer>();
19            PlatformAccount platformAccount = new PlatformAccount();
20
21            foreach (var user in users)
22            {
23                customers.Add(platformAccount.Generate(user));
24            }
25        }
26    }
27 }
```

Figure 4.3: Open–closed example

As seen in the previous screenshot, the routine is creating a list of four users, and for each item, a method **Generate** is from the class

PlatformAccount. This class contains a mapping between user and customer account, as shown in [Figure 4.4](#):

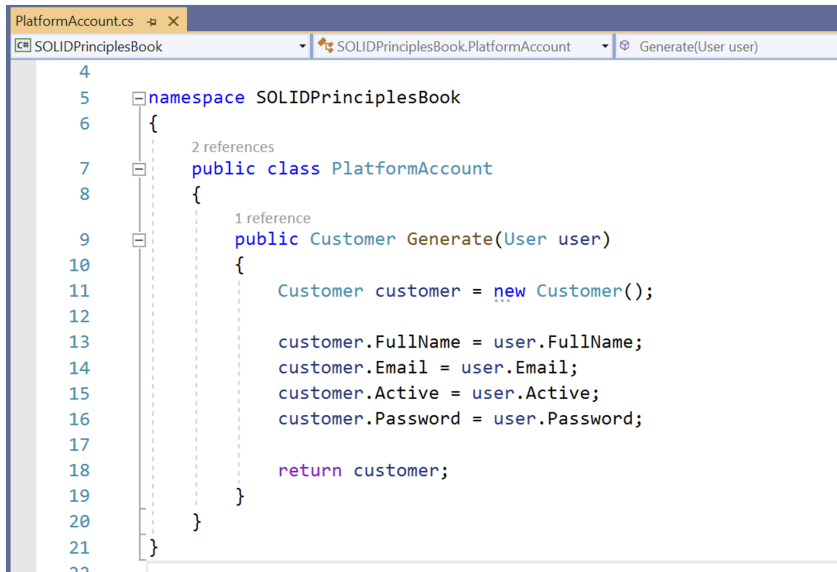


Figure 4.4: Platform account class

The **Customer** and **User** classes in this example have the same fields, as it is shown in the following two code samples:

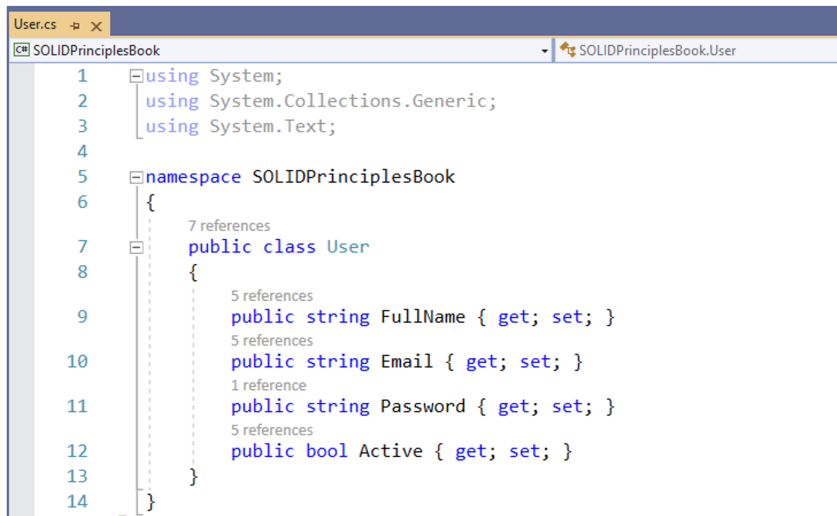


Figure 4.5: User class

The **Customer** class has the representation as shown in the following screenshot:

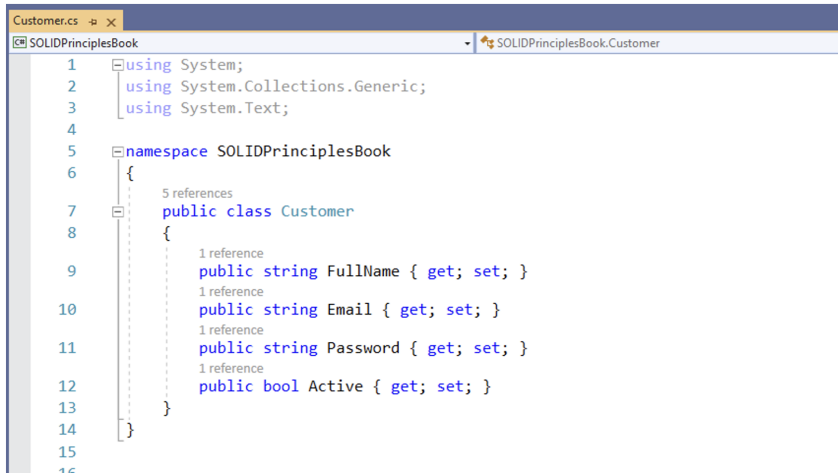


Figure 4.6: Customer class

In real scenarios, the platform account class would use in its implementation a method that would insert the customer in the database and could contain extra code. Still, as the objective is to demonstrate the OCP, the complexity was hidden to facilitate comprehension. Considering that a requirement can change anytime in that part of software because of business requirement modifications, imagine that there is a requirement that the streaming company will start having other types of accounts for customers, such as premium accounts with different logic in comparison to the standard account. In that scenario, there are two main options: changing the implementation of the platform account class to check if the customer has a premium account, and if yes, the method would run a different code, as shown in the following code sample:

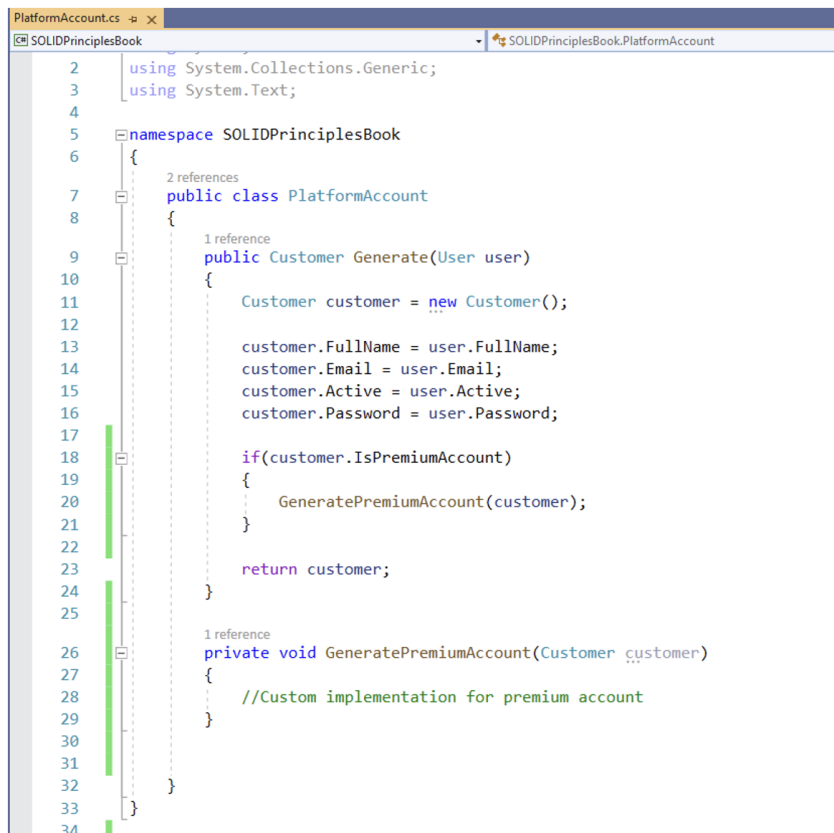


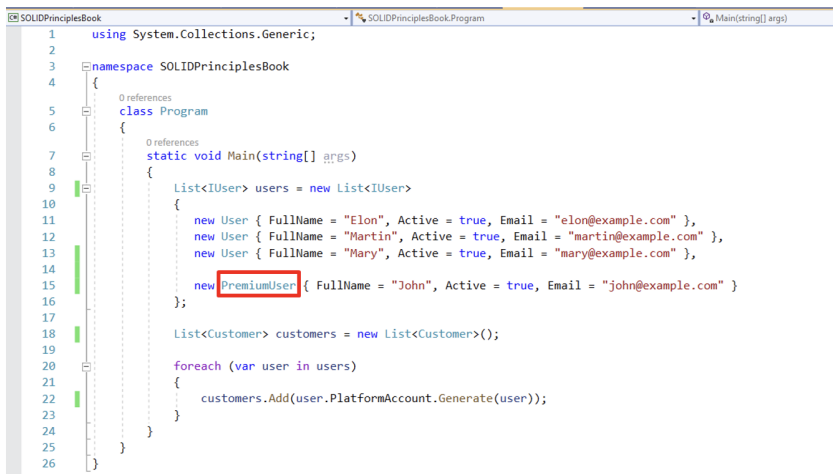
Figure 4.7: Premium account implementation

The implementation, as shown in the previous screenshot, violates the OCP because the platform account class was modified instead of being just extended. It now contains an `if` statement in case the customer has a premium account and it is calling a different method. There are a couple of limitations and risks to using this approach:

- All the methods that have to call the create method must set the property `isPremiumAccount` correctly. It is easy to make mistakes in coding and forget to set the property value to `true`.
- If this class has a complex implementation, adding code to the method could easily affect the behavior for other types of accounts, such as the standard one, that should remain as it is.
- If new account types are included in the future, such as

bronze, silver, and gold, it would add additional conditions to the platform account class and get a higher complexity of the code.

An alternative implementation that follows the recommendations of the OCP would extend the functionality to provide the premium account option instead of changing the existing class used to create the standard account. It is possible to use interfaces to abstract the **User** and **PlatformAccount** classes to meet this target. In that way, the standard and the premium account would do separated classes, and the implementation of the first original class will not be modified, as shown in *Figure 4.8*:



```
1 using System.Collections.Generic;
2
3 namespace SOLIDPrinciplesBook
4 {
5     0 references
6     class Program
7     {
8         0 references
9         static void Main(string[] args)
10        {
11            List<IUser> users = new List<IUser>
12            {
13                new User { FullName = "Elon", Active = true, Email = "elon@example.com" },
14                new User { FullName = "Martin", Active = true, Email = "martin@example.com" },
15                new User { FullName = "Mary", Active = true, Email = "mary@example.com" },
16                new PremiumUser { FullName = "John", Active = true, Email = "john@example.com" }
17            };
18            List<Customer> customers = new List<Customer>();
19
20            foreach (var user in users)
21            {
22                customers.Add(user.PlatformAccount.Generate(user));
23            }
24        }
25    }
26 }
```

Figure 4.8: Premium account implementation

A premium user class was created and shared the same interface as the **User** class, as shown in the subsequent two code samples:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      3 references
7      public class User : IUser
8      {
9          8 references
9          public string FullName { get; set; }
10         8 references
10         public string Email { get; set; }
11         4 references
11         public string Password { get; set; }
12         8 references
12         public bool Active { get; set; }
13
14         3 references
14         public IPlatformAccount PlatformAccount { get; set; } = new PlatformAccount();
15     }
16 }
17
18

```

Figure 4.9: Standard user class

Both of the user classes (**User** and **PremiumUser**) have a property called **PlatformAccount**, which refers to an interface and creates an instance of different classes for the user and user premium objects, as shown in the following example:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      1 reference
7      public class PremiumUser : IUser
8      {
9          8 references
9          public string FullName { get; set; }
10         8 references
10         public string Email { get; set; }
11         4 references
11         public string Password { get; set; }
12         8 references
12         public bool Active { get; set; }
13
14         3 references
14         public IPlatformAccount PlatformAccount { get; set; } = new PremiumPlatformAccount();
15     }
16 }
17
18

```

Figure 4.10: Standard user class

The OCP is highly recommended if the software is already being used in a production environment and new requirements must be added to the existent functionalities. The principle helps to keep an understandable code and voids issues in legacy projects.

The Liskov substitution principle

This principle was defined and created by Barbara Liskov, and the

main point and objective of this good practice are to avoid throwing exceptions in a system when inheritance is not used in a recommended way. Inheritance allows reusing the implementation from a parent and common class across the software. Still, when a specific method or property from the parent class does not apply to all the possible children classes, it indicates that the model does not perfectly reflect the business requirement. It could generate issues in production environments with unhandled exceptions.

To demonstrate this principle, we will create a scenario representing the same conditions the principle tries to solve. For example, imagine a system that allows the creation of user accounts in an online book service, and there are two different types of accounts: premium and standard. The standard account has access to a limited number of titles, and the premium account has unlimited access to the titles and can share access with family members. Both have things in common, and it is natural to create a base class that would be a parent of each one, as shown in [Figure 4.11](#):

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      0 references
8      public class BaseUser
9      {
10         0 references
11         public string FullName { get; set; }
12         0 references
13         public string Email { get; set; }
14         0 references
15         public string Password { get; set; }
16
17         0 references
18         public virtual void GiveAccessFamilyMembers()
19         {
20             Console.WriteLine("Access granted to family members");
21         }
22
23         0 references
24         public virtual void AccessToLimitedTitles()
25         {
26             Console.WriteLine("Access to limited titles");
27         }
28
29         0 references
30         public virtual void AccessToUnlimitedTitles()
31         {
32             Console.WriteLine("Access to unlimited titles");
33         }
34     }
35 }

```

Figure 4.11: User base class

As seen in the previous screenshot, there are, in the **BaseUser** class, properties related to the state of the object, such as full name, email, and password. But, the class also contains methods for giving access to book titles and family members. Considering the standard and premium user classes are derived from the base class, the implementation of both is as *shown in Figure 4.12*:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      public class StandardUser: BaseUser
8      {
9          public override void AccessToLimitedTitles()
10         {
11             base.AccessToLimitedTitles();
12         }
13
14         public override void AccessToUnlimitedTitles()
15         {
16             throw new InvalidOperationException("This type of account does not have unlimited access");
17         }
18
19         public override void GiveAccessFamilyMembers()
20         {
21             throw new InvalidOperationException("This type of account does have access to family members");
22         }
23     }
24
25

```

Figure 4.12: User standard class

The class contains methods that do not necessarily apply to the standard user class but to the premium user class, such as family member access and access to unlimited titles. The problem with this implementation is if an instance of **StandardUser** class is created. By mistake, a method that should only belong to the premium user class is called, the system will throw an exception, and the functionality would miss its integrity. Furthermore, the proposed model does not reflect the business requirements with specific rules for the premium account.

Two ways to solve this problem are to remove the methods related to the premium account from the base class and the other is to use interfaces. The following screenshot represents the first solution:


```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      public class BaseUser
8      {
9          public string FullName { get; set; }
10         public string Email { get; set; }
11         public string Password { get; set; }
12     }
13 }
14
15

```

Figure 4.13: Refactored user base class

The methods were removed from the class because they were not generic to be used by every child, according to the requirements for this example. On the other hand, the methods were accordingly placed in the standard and premium user classes, as shown in the following code sample:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace SOLIDPrinciplesBook
6  {
7      public class StandardUser: BaseUser
8      {
9          public void AccessToLimitedTitles()
10         {
11             Console.WriteLine("Access to limited titles");
12         }
13     }
14
15     public class PremiumUser: BaseUser
16     {
17         public void AccessToUnlimitedTitles()
18         {
19             Console.WriteLine("Access to unlimited titles");
20         }
21
22         public void GiveAccessFamilyMembers()
23         {
24             Console.WriteLine("Access granted to family members");
25         }
26     }
27 }
28

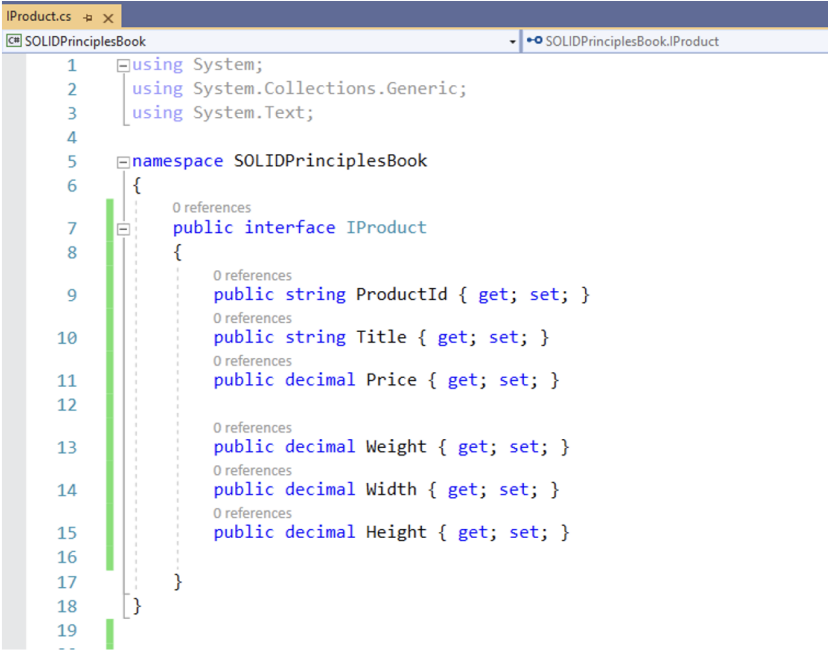
```

Figure 4.14: *Refactored user child classes*

The main point of good inheritance practices in OOP is that the parent class should contain only the generic and standard implementation for the child class. The LSP states that the parent class should be replaceable by the child class and vice-versa. Although the concept of the LSP is slightly different from the OCP, they have similar purposes, which avoid introducing issues in the system when a change is applied based on new requirement requests.

The interface segregation principle

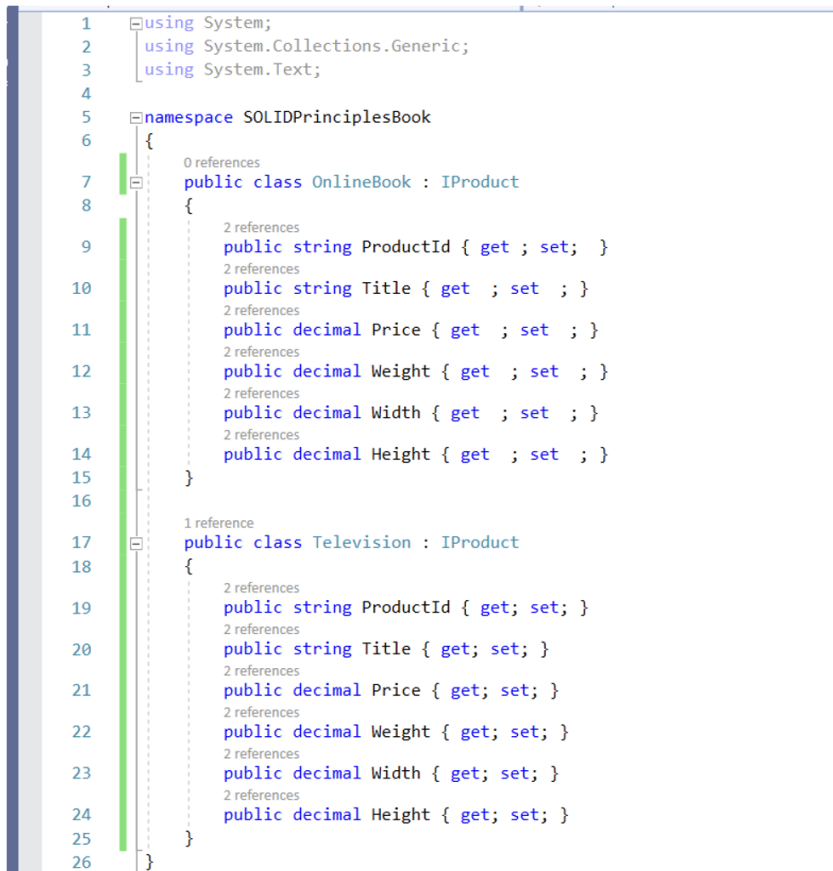
This principle helps the developers to avoid making mistakes using interfaces that do not represent the business requirements. Similar to the LSP, where there is a recommendation of using only the properties and method in a parent class that applies to all child classes, the interface segregation principle applies the same concept regarding interfaces. To clearly understand the principle, refer to the following screenshot that contains an interface for the product:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace SOLIDPrinciplesBook
6 {
7     public interface IProduct
8     {
9         public string ProductId { get; set; }
10        public string Title { get; set; }
11        public decimal Price { get; set; }
12
13        public decimal Weight { get; set; }
14        public decimal Width { get; set; }
15        public decimal Height { get; set; }
16    }
17 }
18
19 --
```

Figure 4.15: *Product interface*

In a common scenario like an online store, the system could easily contain various products with different properties. For instance, the following screenshot represents the implementation of two different classes: **Television** and **OnlineBook**:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace SOLIDPrinciplesBook
6 {
7     0 references
8     public class OnlineBook : IProduct
9     {
10         2 references
11         public string ProductId { get; set; }
12         2 references
13         public string Title { get; set; }
14         2 references
15         public decimal Price { get; set; }
16         2 references
17         public decimal Weight { get; set; }
18         2 references
19         public decimal Width { get; set; }
20         2 references
21         public decimal Height { get; set; }
22     }
23
24     1 reference
25     public class Television : IProduct
26     {
27         2 references
28         public string ProductId { get; set; }
29         2 references
30         public string Title { get; set; }
31         2 references
32         public decimal Price { get; set; }
33         2 references
34         public decimal Weight { get; set; }
35         2 references
36         public decimal Width { get; set; }
37         2 references
38         public decimal Height { get; set; }
39     }
40 }
```

Figure 4.16: Online book and television classes

As shown in the previous screenshot, considering the two classes are implementing the same **IProduct** interface, both have the same properties. The problem with this implementation is that not all types of products share the same properties or methods. For example, the fields related to product dimension (**Width** and **Height**) make sense only for **Television** and not for **OnlineBook**. To have a more precise code definition, the interface segregation principle states that it would be better to create a distinct interface

for each product type, as shown in the following screenshot:

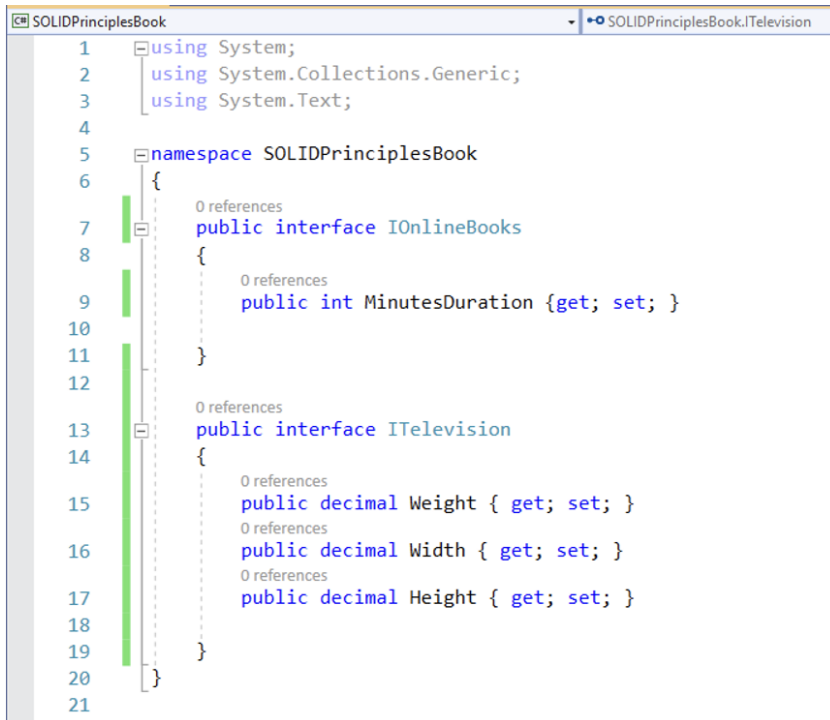


Figure 4.17: Interface segregation

Each specific class must use its underlying interface, which contains the correct fields that should be used in the class. The same concept applies to methods and is one of the most essential best practices in the OOP paradigm.

The dependency inversion principle

The last one of the SOLID principles, the DIP, enforces the excellent practice of hiding the implementation between classes in cases with dependencies. To avoid risks in the development process, each class should contain only the implementation related to the class itself and keep low-level loosely coupled in the relation between them.

For example, a class called **Document** might have a method responsible for converting a document to a PDF or other formats. In real-case scenarios, the implementation of the conversion document

would be placed in a different class, and it would be referred to in the document class, as shown in the following screenshot:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace SOLIDPrinciplesBook
6 {
7     3 references
8     public class Document
9     {
10         private IDocumentConversion PdfConverter = new PdfConverter();
11         private IDocumentConversion ExcelConverter = new ExcelConverter();
12
13         0 references
14         public void GetPdfFormat()
15         {
16             var pdf = this.PdfConverter.Convert(this);
17         }
18
19         0 references
20         public void GetExcelFormat()
21         {
22             var excel = this.ExcelConverter.Convert(this);
23         }
24     }
25 }
```

Figure 4.18: Document class

In this example, there are two apparent dependencies:

- The document class depends on the classes **ExcelConverter** and **PdfConverter**.
- The document class can keep its original functionality for certain operations without directly associating with the converter classes.

Therefore, the document class depends on the other two classes, and the converter classes are dependencies of the **Document** class. Considering that the document conversion process usually uses third-party libraries in case there is a requirement to change the third-party library, the high dependency between the document conversion classes and other parts of the software could be problematic.

An alternative to reduce the dependency between classes and follow the DIP is to pass interfaces as parameters in the constructor of the class that depends on other classes, as shown in the following code sample:

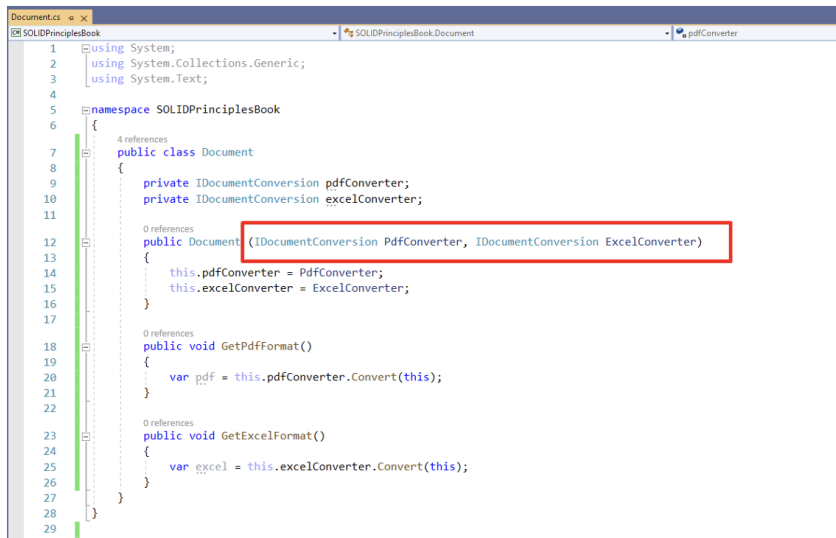


Figure 4.19: *Dependency inversion principle*

Implementing in that way, you can inject the implementation of the converter classes into the **Document** class once they share the same interface. Furthermore, it improves the testability of the classes because it is easier to create mock classes to simulate the behavior of the converter classes.

Conclusion

The SOLID principles were created to help write better code, which applies perfectly to the C# language, considering it is an OOP language. All the principles aim to minimize the impact of changes in existing software and help us write cleaner code. The SRP can be used at the class and method level and even in an architecture design to reduce the dependency between components and to have more reusable functionality. The main point related to OCP and LSP is that it is safer to extend the functionality in the software than change the existing implementation. It avoids break-changes and turns the code more readable. The same benefits apply to the ISP and DIP.

In this chapter, you learned the essential information on SOLID principles. You became familiar with the main advantages of using those principles in different software development scenarios in your

.NET Core projects, creating practical examples.

In the next chapter, you will start the journey across the design pattern with practical examples of the abstract factory pattern using the .NET 5 and C# 9.0 language.

Points to remember

- The OCP is recommended when changes are required in an existing project.
- All the principles have a common objective: to reduce the impact of changes in the software.
- The SOLID principles perfectly apply to the C# language and for projects developed in .NET.

Multiple choice questions

1. **Which SOLID principle states that a class should have only one reason to change?**
 - A. Single Responsibility Principle
 - B. Open/Closed Principle
 - C. Liskov Substitution Principle
 - D. Dependency Inversion Principle
2. **According to the SOLID principles, which principle suggests that higher-level modules should not depend on lower-level modules, but both should depend on abstractions?**
 - A. Single Responsibility Principle
 - B. Open/Closed Principle
 - C. Liskov Substitution Principle
 - D. Dependency Inversion Principle
3. **Which SOLID principle emphasizes that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program?**
 - A. Single Responsibility Principle

- B. Open/Closed Principle
- C. Liskov Substitution Principle
- D. Dependency Inversion Principle

Answers

	A	
	B	
	C	
	D	

Questions

1. Explain the main advantages of the LSP.
2. Which SOLID principle helps us to reduce the dependency between classes?
3. Explain the risks of changes in existing projects.

Key terms

- **Single Responsibility Principle (SRP):** A class should have only one reason to change. This means a class should have only one job or responsibility.
- **Open/Closed Principle (OCP):** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without altering the correctness of the program.
- **Interface Segregation Principle (ISP):** No client should be forced to depend on interfaces they do not use.
- **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. Also, abstractions should not depend on details. Details should depend on abstractions.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the

Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Introduction to Design Patterns

Introduction

With this chapter, we are starting our journey into design pattern concepts, including details on the history of their use in software development, which will help you to identify opportunities while working on a project of software and will drive you to the correct solution for common problems faced in the market in terms of software architecture.

You will learn the importance of design patterns and why they have been used for decades in software development. You will also understand the most common types and categories exemplified in the book's subsequent chapters.

Structure

In this chapter, we will discuss the following topics:

- Why design patterns are important
- The history of design patterns in software development

- The most common design pattern types

Objectives

After studying this unit, you should be able to understand the fundamental concepts of design patterns and identify general categories for common design patterns. In this chapter, we will discuss the use of design patterns in different scenarios in software development.

Importance of design patterns

Design patterns are a popular topic in software development. Still, it is always essential to get the idea of where they originated and why they are so important in our daily routines as software developers, software architects, and any professional who need to build complex software that involves impactful technical decisions.

Common problems in software development can be solved by standard solutions as well, as a relevant number of companies and professionals already have experience in specific scenarios that we can all take advantage of while working on specific projects. Design Patterns are proven solutions for recurring problems in software development.

It has not started in the software development area, but its concepts were borrowed from Christopher Alexander, a civil engineer who documented solutions to common issues related to the construction of buildings. He identified recurring situations in the civil engineering field. Based on his experience in solving problems in these contexts, he not only decided to document the cases but also publish and share them with others.

After the first version of design patterns for civil engineering was published by Christopher Alexander, the idea became famous in other fields, including software engineering, with the specification and creation of the first design patterns for software development after about two decades, which led to the publication of design patterns: Elements of Reusable Object-Oriented Software, back in 1995 by Eric Gamma, Richard Helm, Ralph Johnson, and John Vissides, known as the Gang of Four. This first formal publication about the topic became a considerable influence in the field,

determining 23 patterns that are still useful many years after the fact till these days.

These 23 patterns were chosen because they represented great solutions to known problems in software development, following the same idea of Christopher Alexander. Although these 23 patterns were mentioned in the first publication, hundreds of them were created and documented over time following the evolution and complexity of software development, which includes the need for patterns for a cloud application, distributed systems, data engineering, and more scenarios.

Software projects represent a massive challenge in design and requirements considering their nature is quite abstract and may change during development. It is customary in software engineering to face issues that we never faced before, mainly related to the technical and business aspects of the software. Making the right decisions in software design is extremely important for the success of projects. These decisions significantly impact scalability, maintainability, extensibility, and other vital aspects of good coding techniques and software architecture practices.

Beyond the clear advantage of having patterns to solve common problems in the software industry, the use of Design Patterns helps in the communication between software engineers, setting a common language that can be easily shared in terms of coding structure and details on technical implementation.

History of design patterns

Design patterns are actually from the 1970s after the publication *Pattern Language: Towns, Buildings, Construction*, written by *Christopher Alexander* and other prominent professionals. This publication documented architectural patterns with more than 200 examples. The term *pattern language* referred to the book called *User Centered System Design*, published in 1986, which suggested the use of patterns applied to interaction design for digital projects and services.

In the meantime, extensive studies have started related to software engineering in the 1980s. *Kent Beck* and *Ward Cunningham* used defined design patterns while they were working for Tektronix Test

Systems Group. The ideas of Alexander guided these two professionals in terms of defining a design for the software that would be centered on the user.

At the same time, *Erich Gamma* saw the importance of defining solutions for known problems while working on his Ph.D. thesis. He believed that design patterns would be helpful to improve the quality of projects of software that uses the object-oriented paradigm. As part of the preparation for European conferences on Object-Oriented Programming topics, *Gamma*, *Richard Helm*, and others put a lot of effort into documenting patterns for software development in 1991.

In this conference in 1991, *Gamma* and *Helm* combined their efforts with the work done by *Ralph Johnson* and *John Vissides*, creating an informal group called **Gang of Four (GoF)**, which was a name that became very popular in the software industry. These four professionals published the book *Design Patterns: Elements of Reusable Object-Oriented Software*, which contains 23 patterns, as it was already mentioned in this chapter.

Design patterns did not stop with the first book published by the Gang of Four. Still, it has evolved over time, having a relevant contribution from the company and individual contributors, including the contributions of the Hillside Group. This non-profit organization created the Pattern Languages of Programs, a set of annual conferences with its primary objective to review and evaluate state of the art regarding design patterns. A relevant number of design pattern types are explained in the following sections of this chapter.

Design pattern categories

Design patterns are usually classified into four main categories based on their purpose: creational, structural, behavioral, and concurrency. After the definition of the first 23 design patterns, many others were defined based on new needs in software development, including patterns dedicated to Cloud Architecture and Distributed Systems.

Creational patterns

Creational patterns simplify the creation process of complex objects, separating the logic of building these objects from the original classes themselves. This design pattern usually implements an interface used by multiple classes responsible for building objects with similar structures.

Separating the construction of complex objects allows the separation of the creation into small steps that are easily interpreted by any developer working on the project. Creational design patterns are meant to encapsulate which classes need to be created in runtime, sometimes hiding the instances of these classes from the caller in the application.

It is possible to group creation patterns based on how they are implemented, one being focused on handling how the objects are created and another on how the classes are created. There are five known creation patterns commonly used in the market:

- Abstract factory
- Builder pattern
- Factory method pattern
- Prototype pattern
- Single pattern

In the following chapters of this book, you will have the opportunity to implement all these creational patterns by combining features available in the most recent version of the .NET platform, such as Blazor, GRPC, and others. Despite design patterns being a platform and technology-agnostic topic, this book will show you how to use the most famous patterns to build scalable and extensible applications using the .NET platform and the latest features in the C# language.

Behavioral patterns

Behavioral design patterns are commonly used to implement good practices related to communication between classes, reducing the coupling between them. Some patterns are primarily used in .NET applications, such as mediator and command patterns. The most common behavior patterns are:

- Chain of Responsibility

- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
 - Strategy Pattern
 - Template Method Pattern
 - Visitor Pattern

During the development of enterprise applications exemplified in this book, you will realize that the main objective of this pattern is to pass information between classes in a way that allows the execution of workflows without compromising autonomy and testability for each class.

Understanding the categories behind classifying these patterns is essential to use them properly. Each of them was documented to solve a specific problem, and the classification helps identify the correct pattern to be used in the context of the project we are working on. Any design pattern should be used with criteria to understand the problem that needs to be solved clearly.

Structural patterns

Structural patterns have the purpose of facilitating the relationship between classes and components in one or multiple applications. This pattern is commonly used to establish integration between systems reducing the dependency between the different components. We can see some structural patterns being used mainly in .NET applications, such as the Decorator pattern, and Adapter pattern, among others. The most common structural patterns are:

- Adapter Pattern
- Aggregate Pattern
- Bridge Pattern
- Composite Pattern

- Decorator Pattern
- Extensibility Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

In summary, a structural pattern is an excellent approach to combining multiple objects and classes to consolidate a complex structure to achieve clear goals for the system. The combination of several different components in a system requires great expertise from software engineers and software architects, and a correct understanding of structural patterns is essential to build extensible and flexible applications that are easy to change and maintain over time.

Patterns for distributed systems

The development of complex systems involving multiple applications, databases hosted in multiple regions, and a considerable amount of workload require sophisticated techniques in terms of software architecture to support the system long-term. With the evolution of enterprise applications, the challenges to building distributed became much more complex, requiring the documentation and identification of patterns that would help create distributed systems.

For any application that needs to support simultaneous access by millions of users daily, regardless of the technologies used to build the system, many factors need to be considered in terms of software architecture, such as networking issues, server downtime, synchronization problems, and others.

To solve these recurring problems, software industries, specialists, and big companies documented a series of design patterns exclusively dedicated to distributed systems:

- Clock-Bound Wait
- Consistent Core
- Emergent Leader
- Fixed Partitions

- Low-Water Mark
- Quorum
- Request Batch
- Request Pipeline
- Lease
- Leader and Followers
- HeartBeat
- Version Vector

It is possible to divide the patterns for distributed systems into three categories: object communication, security, and event-driven. The communication category contains patterns to establish a structure for messaging protocols across different components in the system. The security patterns handle integrity, confidentiality, and strong policies for authentication and authorization. Finally, the even-driven patterns focus on the creation, identification, and consumption based on events, largely used in a microservice architecture.

Microservices design patterns

Microservice has been a popular architectural approach for modern applications. Depending on the size of the system, it may involve the development of hundreds of applications, which usually happens with extensive financial systems, e-commerce, and other areas that involve complex transactions and millions of simultaneous requests. Design patterns for microservices address the following challenges: scalability, resiliency, availability, auto-provisioning, failure isolation, and others. Based on these challenges, there are known patterns for microservices that can be applied:

- Decomposition by business capabilities
- Decomposition by subdomain
- Strangler pattern
- API Gateway pattern
- Aggregator pattern
- Database per service

- Saga pattern
- Health Check
- Circuit Breaker Pattern

Many other patterns are used to solve problems in a microservice architecture, including continuous delivery patterns, sidecars, and more. The definition and documentation of patterns is an ongoing process that will never stop. Therefore, the list of patterns for microservice architecture and other purposes will still grow over time.

Conclusion

Correctly understanding design patterns is an essential skill for any software developer. It includes the ability to choose the correct approach to solve complex problems in real projects under the pressure of functional and non-functional requirements involving system performance, scalability, predictability in maintenance, and many other crucial aspects of any software project.

In this chapter, you had the opportunity to learn the history of design patterns applied to software development, the most common categories, with an overview of the importance of design patterns for object-oriented programming, distributed systems, and microservices architecture.

In the next chapter, you will start your practical journey into design patterns by implementing the Singleton Pattern based on real-world examples using .NET 7 and the latest C# version, following good practices for creating enterprise applications.

Points to remember

- Design patterns for software development were inspired by patterns documented for civil engineering, including building architecture.
- Understanding the design pattern categories is recommended to apply each of them to the correct scenarios.
- Design patterns apply not only to object-oriented programming paradigms but to distributed systems, microservices, and many other cases in software development.

- In some instances, developing complex applications will involve a combination of multiple patterns to achieve functional and non-functional requirements.
- Design patterns are essential for any developer who wants to build sophisticated and reliable applications using an object-oriented programming paradigm.

Multiple choice questions

- Which type of design pattern is primarily concerned with object creation mechanisms, aiming to instantiate objects in a manner suitable for the situation?**
 - Creational Patterns
 - Structural Patterns
 - Behavioral Patterns
 - Architectural Patterns
- In an architectural pattern that structures software into interconnected components, which component generally manages the primary user interactions and input?**
 - Creational Patterns
 - Structural Patterns
 - Behavioral Patterns
 - Architectural Patterns
- Which type of design pattern emphasizes determining and implementing common communication patterns amongst objects?**
 - Creational Patterns
 - Structural Patterns
 - Behavioral Patterns
 - Architectural Patterns

Answers

	D	
	S	

Questions

1. Explain the primary purpose of creational patterns.
2. List three patterns related to distributed systems.
3. Describe the main benefits of the use of Design Patterns.

Key terms

- **Creational Patterns:** These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- **Structural Patterns:** Concerned with how classes and objects can be composed, to form larger structures.
- **Behavioral Patterns:** These patterns are about identifying common communication patterns among objects and realize these patterns.
- **Concurrency Patterns:** Focus on multi-threaded programming paradigms. They deal with the multi-threaded execution model in a manner that ensures safe and efficient execution.
- **Architectural Patterns:** These are high-level patterns that deal with software architecture. They provide a set template for the broader system architecture.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Singleton Pattern in .NET Applications

Introduction

Many scenarios in software development require keeping a single instance of a particular object to be shared across the entire system for a well-defined purpose, such as sharing database connections, in-memory caching objects, or an instance of a class that is responsible for keeping a consistent state for all parts of the application. The Single Pattern allows us to achieve this objective, being one of the most used and known patterns in .NET applications using dependency injection.

Learning the single pattern allows you to understand in which scenarios the use of this pattern can be helpful. It also helps you recognize the pattern in legacy projects and third-party libraries.

Structure

In this chapter, we will discuss the following topics:

- Single Pattern concept

- Examples in C# and .NET
- Performance enhancements for dependency injection

Objectives

After studying this unit, you should be able to understand the Single Pattern, apply it in real-world projects, and identify the use of this pattern in legacy projects and third-party libraries.

Single pattern definition

The single pattern, as the name suggests, has as its primary objective to guarantee that only one instance of a specific object is available in the application, sharing the same state of this object globally for other classes and components of the software. With this pattern, the structure of the underlying class restricts the creation of other instances of objects, usually using the static modifier. The .NET platform introduced many improvements regarding the use of single pattern since the .NET Core 1.0 version, combining the use of this pattern with the concept of dependency injection, becoming relevantly easy to manage and keep a single point of access to a specific object in the memory without the need to force the use of the pattern itself programmatically.

Like other patterns, the singleton pattern has advantages and disadvantages depending on its applied scenarios. One of the benefits is the certainty that the system will safely keep the same object for the entire system, managing possible concurrence challenges correctly. Following the best practices of object-oriented programming, the use of this pattern restricts the specification of the corresponding classes with private constructor methods, which avoids the possibility of violation and prevents the creation of inappropriate multiple instances of objects with distinct states.

In terms of negative points, the single pattern turns the task of implementing unit tests much more complex once the classes involved are marked with the static modifier, being a challenge to mock the state for verification in terms of application execution behavior. Additionally, considering this pattern has the explicit purpose of keeping a single global access point to a specific object for the entire application, concurrent and parallel threads accessing

the objects need to take of preventing locks from guaranteeing that all the requests to the object receive the same result, which may cause performance issues in critical systems where any extra nanoseconds in the response performance are crucial.

Considering you have learned the theoretical concepts of the singleton pattern, the next section will show you how to apply the pattern in distinct scenarios in a hands-on approach, and it will allow you to understand how the Singleton Pattern is used natively in distinct project types in the .NET platform.

Single pattern definition

As each design pattern has a defined purpose of solving a specific problem in software development using sound practices of OOP and software architecture, it is crucial to learn any pattern building a scenario where real problems in software development need to be solved by each pattern. In the context of this chapter, the single pattern itself.

Given that, imagine a hypothetical scenario where a system must keep the same instance of an object globally, which is responsible for communicating with an external resource, such as third-party Web services that convert documents to PDF format sob demand or connection to a remote database. One of the requirements in the scenario would be to restrict the number of authentication processes that happened to the external service, limiting it to a single one, avoiding extra costs and performance issues for the whole process. Therefore, a single session for the authentication to the PDF Converter service must remain open as much as possible, being shared between all the application routines that rely on the PDF Converter service. If a standard class is used in this case, the class would have an implementation similar to the one shown in [Figure 6.1](#):


```

public class PDFConverter
{
    private Session _session;
    private PDFService _pdfService;

    0 references
    public PDFConverter()
    {
        _pdfService = new PDFService();
        _session = _pdfService.StartSession();
    }

    0 references
    public void ConvertToPDF(string filePath)
    {
        _pdfService.Convert(filePath, _session);
    }
}

```

Figure 6.1: PDF converter class

Once the PDF Service class requires a **Session** object in the converter method, as seen in the **ConvertToPdf** method body, a new Session object is created each time a new instance of the class **PDFConverter** is generated. Considering there is a requirement in the given scenario that states that a single Session should be used as much as possible for multiple calls to the external service, the current implementation does not perfectly satisfy the requirements because multiple instances of the class can be created without any restriction with this version of the code. To see what is the output for the given code, a new method called **GetSessionInfo** can be implemented in the class, as shown in [Figure 6.2](#):

```

21  0 references
22  public void GetSessionInfo()
23  {
24      Console.WriteLine("Session Info: " + _session.Id);
}

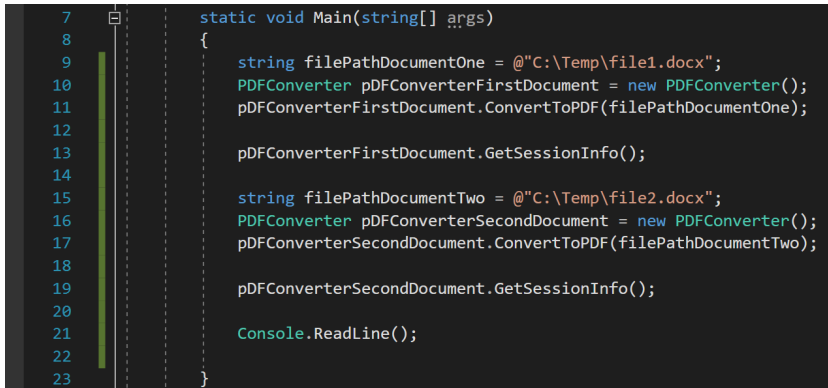
```

Figure 6.2: Get session info method

The information displayed by the method allows us to follow the workflow execution in runtime while testing the class to verify that the entire application uses different instances of the session of the object each time.

Multiple instances

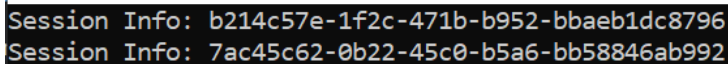
If we place the code from [Figure 6.1](#) and [Figure 6.2](#) in a regular Console application, we can call the **PDFConverter** class multiple times in the **Program.cs** file to verify if the application does not correctly handle the session object creation by generating multiple instances. You can adapt the **Main** method in the **Program.cs** file using the following code shown in [Figure 6.3](#):



```
7 static void Main(string[] args)
8 {
9     string filePathDocumentOne = @"C:\Temp\file1.docx";
10    PDFConverter pdfConverterFirstDocument = new PDFConverter();
11    pdfConverterFirstDocument.ConvertToPDF(filePathDocumentOne);
12
13    pdfConverterFirstDocument.GetSessionInfo();
14
15    string filePathDocumentTwo = @"C:\Temp\file2.docx";
16    PDFConverter pdfConverterSecondDocument = new PDFConverter();
17    pdfConverterSecondDocument.ConvertToPDF(filePathDocumentTwo);
18
19    pdfConverterSecondDocument.GetSessionInfo();
20
21    Console.ReadLine();
22 }
23
```

Figure 6.3: Multiple PDF converter instances

Between lines nine and eleven, the first document is converted to PDF using one of the instances of the PDF Converter class, and between lines fifteen and seventeen, the second document is converted using a new instance of the converter, a different one. As the program calls the **GetSessionInfo** method twice, in lines thirteen and nineteen), when the Console application runs, the system shows on the screen the Session information for the two document conversions, as shown in [Figure 6.4](#):



```
Session Info: b214c57e-1f2c-471b-b952-bbaeb1dc8796
Session Info: 7ac45c62-0b22-45c0-b5a6-bb58846ab992
```

Figure 6.4: Session information display

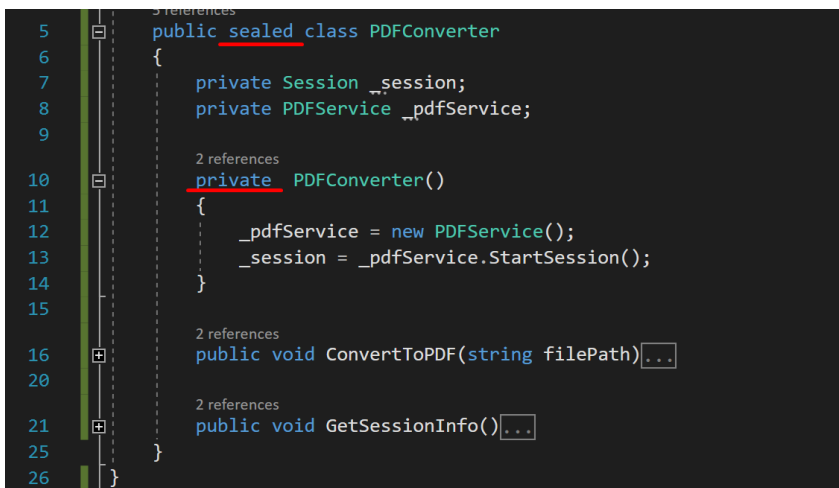
As shown in [Figure 6.4](#), two different sessions are generated, which means that the implementation does meet the desired requirement. Despite this is just an example of creating multiple instances in the

same method in the `Program.cs` file within a Console application, the idea stated by the requirements in the scenario is to create an architecture in the code that forces the entire application to have global access to a single instance of the `PDF Converter` object, with the purpose of avoiding the possibility of making mistakes in the development. Therefore, the software design must have a structure that explicitly restricts the creation of multiple instances of the `PDF Converter` class. That represents a common problem in software development, as it is not conceivable to restrict the violation of a requirement without using a mechanism that denies the creation of multiple instances of an object.

Modifications for the single pattern

The single pattern is a simple and sophisticated solution for the scenario given in this chapter for learning purposes, as its implementation requires significant changes in the `PDFConverter` class to prevent the creation of multiple instances and to guarantee that the application would not face any issues in regards to concurrent access to the external resource responsible for converting documents to PDF.

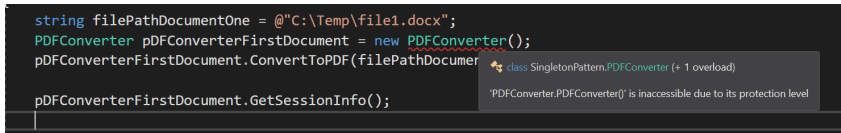
First of all, it is crucial to understand that all the class constructors in the single pattern should use the private modifier, and the underlying classes should use the sealed modifier, as highlighted in [Figure 6.5](#):



```
5 public sealed class PDFConverter
6 {
7     private Session _session;
8     private PDFService _pdfService;
9
10    2 references
11    private PDFConverter()
12    {
13        _pdfService = new PDFService();
14        _session = _pdfService.StartSession();
15    }
16
17    2 references
18    public void ConvertToPDF(string filePath)...
19
20
21    2 references
22    public void GetSessionInfo()...
23
24 }
25
26 }
```

Figure 6.5: First modifications of the PDF Converter class

The sealed modifier prevents other classes from inheriting from the **PDFConverter** class, a necessary restriction to avoid creating multiple instances of objects via inheritance. However, after these small changes to the class, the routine in the **Program.cs** file shows an expected error, considering it is not allowed to create a new instance of the **PDF Converter** class after we have modified the constructor visibility with the **private** modifier, as shown in [Figure 6.6](#):



```
string filePathDocumentOne = @"C:\Temp\file1.docx";
PDFConverter pdfConverterFirstDocument = new PDFConverter();
pdfConverterFirstDocument.ConvertToPDF(filePathDocumentOne);
pdfConverterFirstDocument.GetSessionInfo();
```

class SingletonPattern.PDFConverter (+ 1 overload)

'PDFConverter.PDFConverter()' is inaccessible due to its protection level

Figure 6.6: Compile error

The actual objective of the single pattern is to avoid the creation of multiple instances of an object, but after the changes to the **PDFConverter** class, it is not possible anymore to create any instance of the **PDFConverter** type, which means that the implementation of the single pattern was not achieved yet. By convention, the single pattern requires to use of a **GetInstance** static property, which handles the creation of a new instance of the object only if the state of the object is **null**, as highlighted in [Figure 6.7](#):

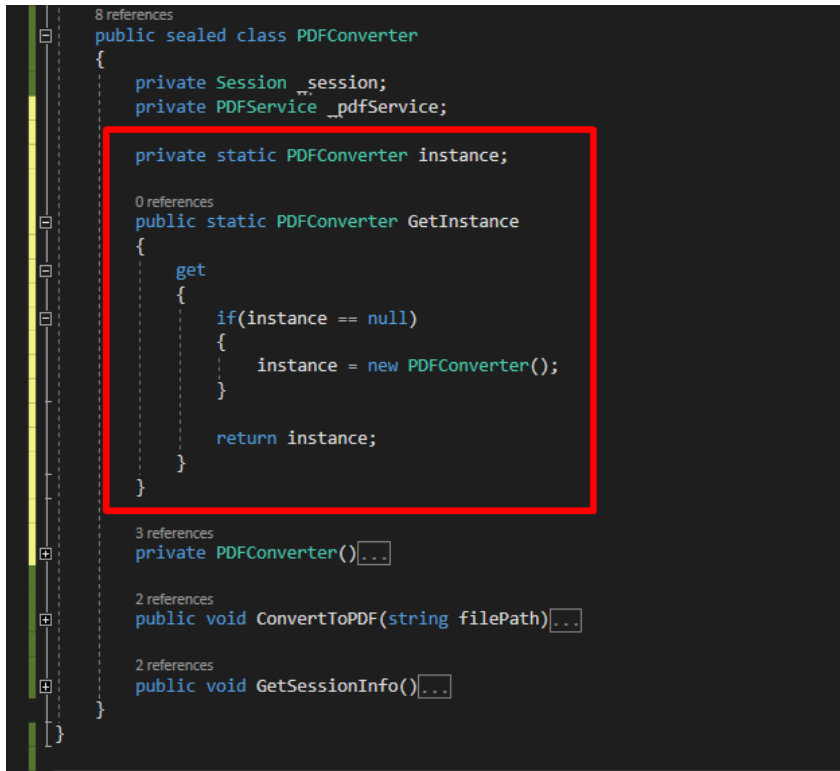


Figure 6.7: Get Instance method

In this case, the class constructor method is still private, and the class now has a static method that always returns the same object instance except when the object is null. That implementation guarantees that the same instance will be used across the entire application, even if the class is referred to several times in the program.

After these changes, the principal requirement is satisfied in terms of coding design and implementation, requiring small extra changes in the **Program.cs** file. Considering a static property is being used by the **PDFConverter** class, the program keeps only one reference in the memory for the object, preventing the possibility of creating new instances. Although the result is entirely different, the code remains relatively similar, as shown in [Figure 6.8](#):

```

7      static void Main(string[] args)
8      {
9          string filePathDocumentOne = @"C:\Temp\file1.docx";
10         PDFConverter pDFConverterFirstDocument = PDFConverter.GetInstance;
11         pDFConverterFirstDocument.ConvertToPDF(filePathDocumentOne);
12
13         pDFConverterFirstDocument.GetSessionInfo();
14
15         string filePathDocumentTwo = @"C:\Temp\file2.docx";
16         PDFConverter pDFConverterSecondDocument = PDFConverter.GetInstance;
17         pDFConverterSecondDocument.ConvertToPDF(filePathDocumentTwo);
18
19         pDFConverterSecondDocument.GetSessionInfo();
20
21         Console.ReadLine();
22     }
23

```

Figure 6.8: Changes in the Program.cs file

Instead of calling the class constructor method, the static property is called for the **PDFConverter** class type object. Therefore, the routine present in the **Program.cs** file had a tiny impact in terms of implementation. If we run the Console application, both document conversions use the same session, as shown in [Figure 6.9](#):

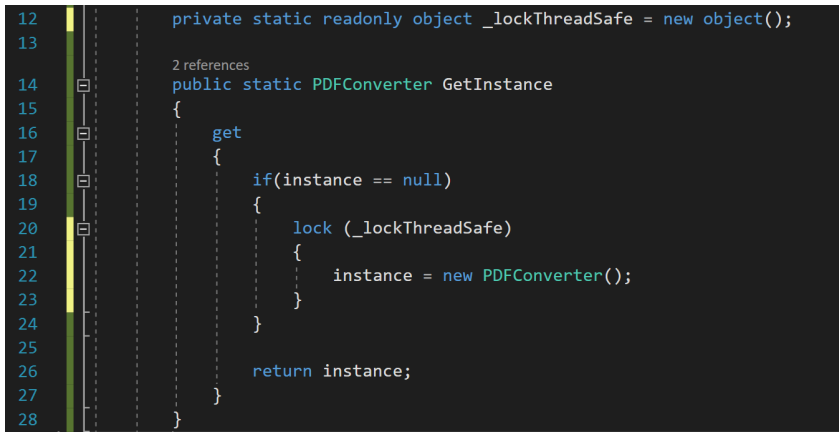
```

Session Info: d5494576-dcf8-4c26-af15-3d27bfc401dd
Session Info: d5494576-dcf8-4c26-af15-3d27bfc401dd

```

Figure 6.9: Session result

The current single pattern implementation presents limitations in scenarios where there is a considerable amount of simultaneous requests to the **PDFConverter** instance, as it may cause concurrency problems in the attempt to get information from memory, causing locks to the thread. To avoid errors, in this case, it is safer to use the lock feature existing in C# language, which will prevent the same resource from being accessed simultaneously for the block of the code responsible for managing the creation of the single instance of the **PDFConverter** object. All the other threads must wait to access the resource with the lock statement. The following code demonstrated in [Figure 6.10](#) shows a change to the class with the thread-safe implementation:



```

12 private static readonly object _lockThreadSafe = new object();
13
14 public static PDFConverter GetInstance
15 {
16     get
17     {
18         if(instance == null)
19         {
20             lock (_lockThreadSafe)
21             {
22                 instance = new PDFConverter();
23             }
24         }
25         return instance;
26     }
27 }
28

```

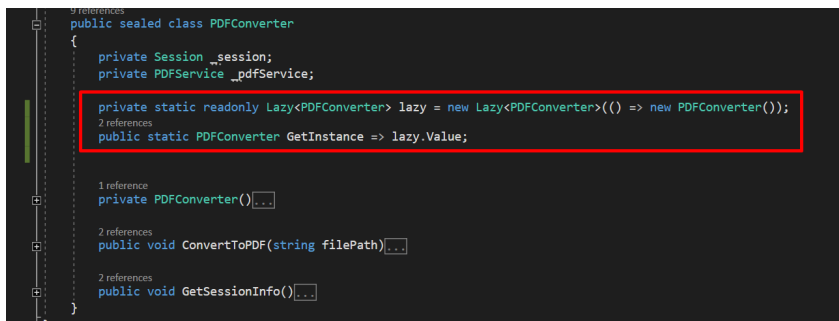
Figure 6.10: Lock statement

The use of a `lock` statement, in this case, is recommended in high-demanded scenarios. However, this feature needs to be used carefully as it may cause performance problems, as the application might have a relevant number of requests in the queue to access a specific resource.

Thread-safe implementation

A private property called `lockThreadSafe` is used in the new version of the code in [Figure 6.10](#), and the `GetInstance` static property locks that properly before creating a new instance of the `PDFConverter` class in the cases where it is null. Usually, this implementation is called **Thread Safety Singleton**, which represents an excellent alternative where intense concurrent access to a specific routing might cause errors in the application.

A different approach that can be taken regarding the single pattern is using the **Lazy** type in C# language, a feature available that allows us to use the concept of **thread-safe** implementation implicitly without adding the extra lock statement in the code. If we refactor the previous code to use the **Lazy** type, the implementation looks like the code presented in [Figure 6.11](#):



```

9 references
public sealed class PDFConverter
{
    private Session _session;
    private PDFService _pdfService;

    private static readonly Lazy<PDFConverter> lazy = new Lazy<PDFConverter>(() => new PDFConverter());
    2 references
    public static PDFConverter GetInstance => lazy.Value;

    1 reference
    private PDFConverter()...

    2 references
    public void ConvertToPDF(string filePath)...

    2 references
    public void GetSessionInfo()...
}

```

Figure 6.11: Lazy type

Using the **Lazy** type reduces the amount of code needed to implement the single pattern and produces the same result, a great alternative to having a more readable code. The .NET platform provides an extra option to handle the single pattern in ASP.NET Core applications and other types, which consists of the encapsulation of the process to keep a single instance of an object for the entire application, using Dependency Injection that is explained in the next section.

Dependency injection

The .NET platform, since the version .NET Core 1.0, introduced the possibility of working with Dependency Injection without having to install any extra third-party libraries to achieve this goal. This feature allows us to take the benefits of the single pattern preserving the original implementation of the class, which facilitates the creation of unit tests as the original classes do not need to use static properties to keep a single instance of an object in the memory.

To check how Dependency Injection can be used in a .NET application, create a new ASP.NET Core MVC project, and create the **PDFConverter** class in the project, as seen in [Figure 6.12](#):


```

5 | public class PDFConverter : IPdfConverter
6 | {
7 |     private Session _session;
8 |     private PDFService _pdfService;
9 |     0 references
10 | public PDFConverter()
11 | {
12 |     _pdfService = new PDFService();
13 |     _session = _pdfService.StartSession();
14 | }
15 |
16 | 3 references
17 | public void ConvertToPDF(string filePath)...
18 |
19 |
20 | 3 references
21 | public string GetSessionInfo()
22 | {
23 |     return "Session Info: " + _session.Id;
24 | }

```

Figure 6.12: PDF Converter class for the ASP.NET MVC project

As you can see in [Figure 6.13](#), the class implements the **IPdfConverter** interface, which can be represented as shown in the following:

```

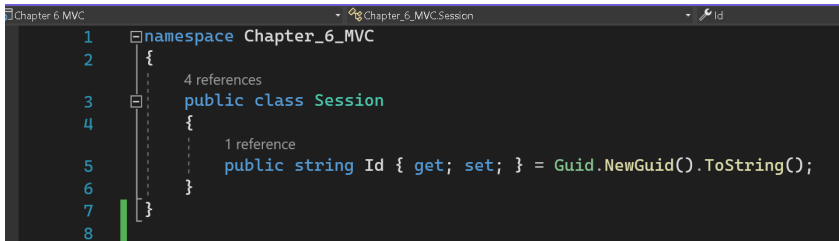
IPdfConverter.cs
Chapter 6 MVC
1 | namespace Chapter_6_MVC
2 | {
3 |     4 references
4 |     public interface IPdfConverter
5 |     {
6 |         3 references
7 |         void ConvertToPDF(string filePath);
8 |         3 references
9 |         string GetSessionInfo();
10 |     }

```

Figure 6.13: IPdfConverter interface

The use of interfaces is essential to use Dependency Injection in .NET applications, as it facilitates the use of different types of implementation for the interfaces based on the application needs without having to change several parts of the application that have reference to the interface.

The PDF Converter also references the **Session** class, which is responsible for generating the session ID used by the external service. For studying purposes, the implementation of the **Session** class looks like the code sample in [Figure 6.14](#):

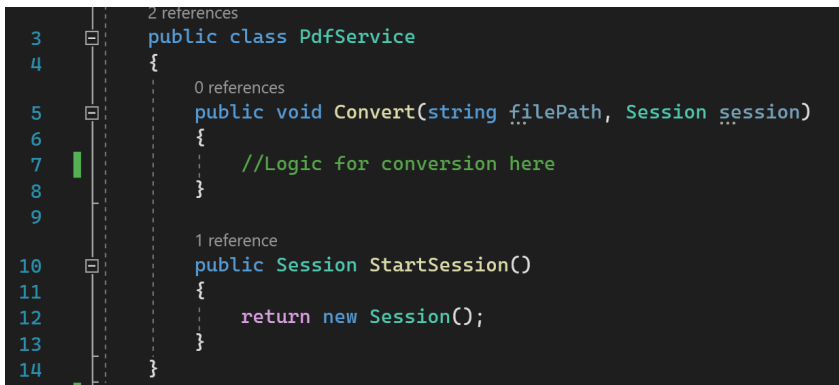
The image shows a code editor window with the title 'Chapter 6 MVC' and 'Chapter_6_MVCSession'. The code is for a C# class named 'Session' inside the 'Chapter_6_MVC' namespace. The class has a single property 'Id' of type 'string' with a getter and setter. The setter is assigned the value 'Guid.NewGuid().ToString()'. There are 4 references to the class and 1 reference to the 'Id' property. The code is as follows:

```
1 namespace Chapter_6_MVC
2 {
3     4 references
4     public class Session
5     {
6         1 reference
7         public string Id { get; set; } = Guid.NewGuid().ToString();
8     }
9 }
```

Figure 6.14: Session class

The **Session** class generates a new ID value when a new class instance is created based on the **Guid** class. This implementation is helpful for us to test the single pattern properly and Dependency injection as the **PDFConverter** class has a private property of the **Session** type. We want to ensure that the same object instance is shared across the application for multiple requests.

Additionally, the **PdfService** class has a reference to the **PdfService** class, which is represented in [Figure 6.15](#):

The image shows a code editor window with the title 'Chapter 6 MVC' and 'Chapter_6_MVCSession'. The code is for a C# class named 'PdfService'. The class has a single method 'Convert' that takes a 'string filePath' and a 'Session session' as parameters. The method body contains a comment '//Logic for conversion here'. There is also a 'StartSession' method that returns a new 'Session' object. There are 2 references to the class and 1 reference to the 'Session' class. The code is as follows:

```
3     2 references
4     public class PdfService
5     {
6         0 references
7         public void Convert(string filePath, Session session)
8         {
9             //Logic for conversion here
10        }
11
12        1 reference
13        public Session StartSession()
14        {
15            return new Session();
16        }
17    }
```

Figure 6.15: PDF Service class

For this ASP.NET MVC project, the framework's responsibility is to handle the single pattern strategy for the **PDFConverter** class using dependency injection. For any class or object that needs to share the

same instances across the application or inject it automatically into different classes, you must register it in the services configuration for the .NET application, which is usually achieved in the **Program.cs** file, where it is possible to register classes and interfaces with the Singleton state, as shown in [Figure 6.16](#):

```
6
7
8     var builder = WebApplication.CreateBuilder(args);
9
10    // Add services to the container.
11    builder.Services.AddControllersWithViews();
12
13    //Dependency Injection for PDF Converter
14    builder.Services.AddSingleton<IPdfConverter, PdfConverter>();
15
16    var app = builder.Build();
```

Figure 6.16: Service configuration for PDF converter

In line thirteen, a service is registered using the **AddSingleton** method and passing the **PdfConverter** interface and the corresponding concrete class. In all the classes that the **PdfConverter** needs to be used, you need to allow injection of the object via a constructor, as shown in [Figure 6.17](#) for the **HomeController**:

```
References
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IPdfConverter _pdfConverter;

    0 references
    public HomeController(ILogger<HomeController> logger,
        IPdfConverter pdfConverter)
    {
        _logger = logger;
        _pdfConverter = pdfConverter;
    }
}
```

Figure 6.17: Injection for HomeController

By applying these modifications to the **Program.cs** and **HomeController.cs** files, all the requests the **Home Controller** will receive the same instance of the **PDF Converter** object without having to handle the singleton pattern programmatically. To verify if the Dependency Injection is working correctly if you can modify the **Index** Action in the controller using the code shown in

Figure 6.18:

```
public IActionResult Index()
{
    List<string> sessionInfoList = new List<string>();

    string filePathDocumentOne = @"C:\Temp\file1.docx";
    _pdfConverter.ConvertToPDF(filePathDocumentOne);

    sessionInfoList.Add(_pdfConverter.GetSessionInfo());

    string filePathDocumentTwo = @"C:\Temp\file2.docx";
    _pdfConverter.ConvertToPDF(filePathDocumentTwo);

    sessionInfoList.Add(_pdfConverter.GetSessionInfo());

    return View(sessionInfoList);
}
```

Figure 6.18: Index Action

The **Index** action converts two documents to PDF using the **PdfConverter** object, sharing the same object. It populates a list of string with session ID information, passing the value to the View. To represent the session information visually, implement the following code in the **Index.cshtml** file, as shown in [Figure 6.19](#):

```
Index.cshtml  X
1  @model List<string>
2
3  @{
4      ViewData["Title"] = "Home Page";
5  }
6
7  <div class="text-center">
8      @foreach (var session in Model)
9      {
10         <h1>@session</h1><br />
11     }
12 </div>
13
```

Figure 6.19: Index view

As shown in [Figure 6.18](#), the constructor of the `HomeController` class receives a `PDFConverter` interface, and the application automatically injects an instance of the `PdfConverter` object to be used every time there is a new request to the controller. The `Index` action has a similar implementation that was done for the Console application.

Still, it stores a list of string objects with the session information for two calls of the `PdfConverter` class. If the application runs multiple times, the system will keep the same session ID for every call, even if the `Index` page is refreshed multiple times. After changing the `Index` view to show the session list, the result is similar to the previous implementation of the singleton pattern for the Console application, with the difference that the state is handled by the ASP.NET Core application implicitly. If you run the application, you will be able to verify that the correct behavior is achieved, as seen in [Figure 6.20](#):

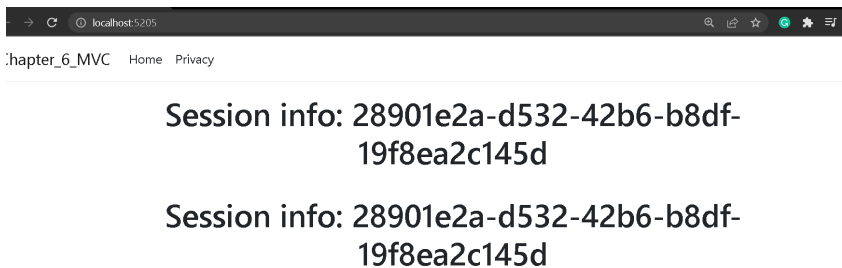


Figure 6.20: *Index view with session information*

Even the `PdfConverter` class does not implement the Singleton Pattern explicitly, considering the class is registered in the services configuration for the .NET application to behave as Singleton, the result satisfies the requirement, keeping a single instance of the object across the application, in all the places where the class is being injected in the constructor. As shown in [Figure 6.20](#), the same session is shared between two calls, even if the page is refreshed several times. A new session ID will be generated only when the application restarts in the server or if, for some reason, the state of the object is recycled in the server memory.

Conclusion

As you learned in this chapter, the Singleton Pattern helps maintain the same instance of an object across the application, solving a common problem in software development in the cases where a system needs to provide a single point of access to a specific resource with the exact same state. This pattern has been largely used in .NET applications since .NET Core 1.0, significantly improving until .NET 7. Dependency Injection introduced a non-invasive way to work with the Singleton Pattern, preventing the need to create static properties for classes, allowing us to write unit tests, and keeping the original classes safe from technical changes.

With this chapter, you had the opportunity to learn how to implement the single pattern in real projects using the C# language and understand which scenarios the pattern can be used. Moreover, you have learned how to apply the pattern in the ASP.NET Core application using Dependency Injection.

In the next chapter, you will have the chance to continue your journey into design patterns using C# language and the .NET platform, learning the Abstract Factory Pattern in combination with Blazor applications.

Points to remember

- The Singleton Pattern can be manually implemented using a private constructor and static properties in the class.
- The lock statement can be used to prevent concurrence issues in the Singleton Pattern implementation.
- The .NET platform allows us to use the Singleton Pattern with the registration of services in the Program class to achieve the objective of having a unique instance of an object across the application using Dependency Injection.

Multiple choice questions

1. Which feature in the C# language can be used to avoid issues in concurrent access to a resource?
A. Lock statement
B. Factory

- C. Static
 - D. Private
2. In which class a singleton class can be registered in the ASP.NET Core applications?
- A. Program class
 - B. Global.asax
 - C. Controllers
 - D. View
3. What is the alternative to the lock statement for singleton classes?
- A. Tasks
 - B. Lazy type
 - C. Parallel loops
 - D. None of the alternatives

Answer

	A	
	B	
	C	
	D	

Questions

1. Explain the primary purpose of the singleton pattern.
2. What are the main benefits of registering the class as Singleton in the services of the ASP.NET applications?
3. According to what you have learned in this chapter, create a hypothetical scenario where the singleton pattern could be applied apart from the example given in this chapter.

Key terms

- **Single Instance:** It is the core idea behind the Singleton pattern. The pattern ensures that a class has only one instance

throughout the lifecycle of the application, preventing any mechanism from creating more than one instance.

- **Global Access Point:** Beyond ensuring a single instance, the Singleton pattern also provides a global point of access to this instance. This ensures that every part of the software can access the singleton instance without needing to instantiate it directly.
- **Lazy Initialization:** This is a technique where the singleton instance is not created until it is needed. Lazy initialization is particularly useful when the creation of the instance is resource-intensive, and it's not necessary until it's specifically requested.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Abstract Factory Pattern with Blazor

Introduction

A robust software architecture design represents a considerable challenge in applying advanced concepts of the object-oriented programming paradigm, including a correct definition of abstractions that faithfully correspond to complex business requirements presented in real scenarios. There are numerous solutions based on the level of complexity that needs to be applied when software is built, and in this chapter, you will have the opportunity to understand the Abstract Factory Pattern, an important approach to simplify the creation of complex objects.

Learning the Abstract Factory Pattern allows you to elegantly apply the best practices of object-oriented programming regarding polymorphism and encapsulation while developing a stable structure to extend and scale the complexity of a project that quickly supports requirement changes. Knowing this design pattern is crucial to familiarize yourself with other creational patterns used in the market.

With this chapter, you will have the opportunity to understand the Abstract Factory Pattern, applying its concept in a step-by-step approach for a Blazor application.

Structure

In this chapter, we will discuss the following topics:

- Abstract Factory Pattern concept
- Examples in C# and .NET
- Practical implementation of Abstract Factory in Blazor applications

Objectives

After studying this unit, you should be able to understand the Abstract Factory Pattern, apply the Abstract Factory Pattern in real-world projects, and identify the use of encapsulation, polymorphism, and other object-oriented programming concepts.

Abstract Factory definition

Any design pattern in software development represents a general solution for common situations that frequently happen in recurring scenarios, being a robust alternative to building a software architecture that follows the most recognized approach used widely in the market. All software projects vary in business and non-functional requirements, but many have similar technical challenges.

For instance, imagine a scenario where a system has the object of allowing users to register products and services for an online store. Different companies sell different types of products, but the infrastructure behind the implementation of this kind of system is quite similar in terms of classes, business logic, data models, database structure, and technical requirements, including the need to control transactions and support the registration of a considerable amount of entries that contain different configuration in terms of properties and characteristics.

Therefore, it is possible to say that there is a pattern followed by

any online store, which requires building and supporting the creation and management of complex records, many families of products, and the ability to provide a particular behavior of the system in terms of business requirements and workflow execution based on each product type.

In this context, among the list of all the available design patterns available, there is the Abstract Factory Pattern, which allows us to create families of related objects, abstracting the complexity of the creation of those from the part of the application that is consuming these objects. In summary, in the case where the system needs to create objects that contain distinct ways to create their corresponding properties based on sophisticated rules, the Abstract Factory Pattern encapsulates the implementation and the logic behind the construction of objects from the concrete classes, giving us the flexibility of extending the software architecture by inserting more types of objects of the same family, without compromising the integrity of the implementation and consistency in the application.

As demonstrated previously in this book, the correct use of interfaces in software development based on the object-oriented programming paradigm is one of the essential characteristics of having a testable, extensible, and reliable structure, representing one of the critical points of the Abstract Factory Pattern.

Abstract Factory scenario

Considering the Abstract Factory Pattern reduces the complexity of the creation families of objects, it is essential to apply its concepts in projects that have requirements similar to the problems that the Abstract Factory Pattern is meant to solve. Therefore, in this section, you will have the opportunity to implement the base structure of a mobile company that sells various types of plans to customers related to text messages, mobile data, and other services, which represents a scenario that the Abstract Factory Pattern can solve with the following requirements:

- The company will offer two types of plans for customers: pre-paid and post-paid
- Both plans must have different conditions for text messages, internet connection speed, and mobile data limits

- The pre-paid option has a limited plan for text messages, up to 1,000 messages monthly
- The pre-paid option has a maximum speed of 10 megabytes per second
- The pre-paid plan has a 10 gigabytes limit for data transfer within the same month
- The postpaid option offers unlimited data transfer and an unlimited number of text messages
- The postpaid plan has a high-speed internet connection, up to 500 megabytes per second

The requirements clearly show that the mobile plans have the same structure in terms of properties for the objects that would be defined in the application using C# language, but the type and the state of the objects are slightly distinct, as shown in [Table 7.1](#):

	Table 7.1: Mobile plans			
Pre-paid	Post-paid	Pre-paid	Post-paid	Pre-paid
10000 bytes/month				
100 megabytes/sec				

Table 7.1: Mobile plans

Given that scenario, the use of the Abstract Factory Pattern allows us to build these complex objects by abstracting their implementation from the highest layer of the application, which is in this pattern, the **Client** layer, as shown in [Figure 7.1](#):

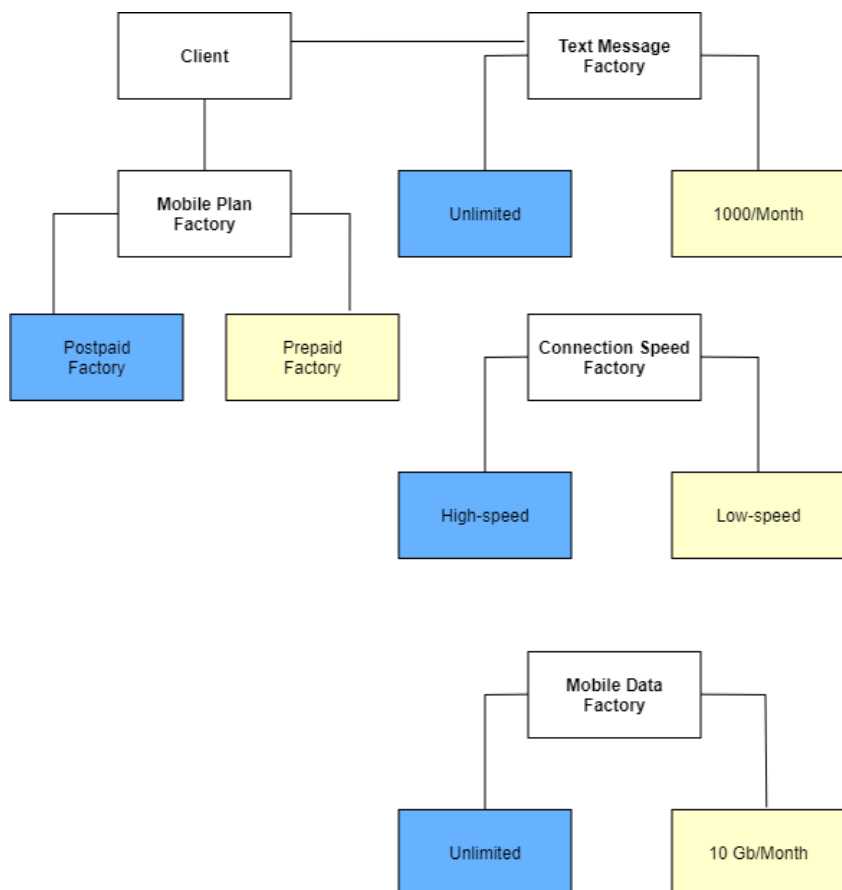


Figure 7.1: Abstract Factory implementation

The correlation between the mobile plan and the type of compound objects is highlighted in [Figure 7.1](#), with blue for the postpaid plan and yellow for the pre-paid plan. The Abstract Factory is known in the market as the factory of factories pattern, once the subset of objects created by the main factory is other factories.

Abstract Factory implementation

Considering the requirements and model structure established in the given scenario, the first step for implementing the Abstract Factory pattern is to create individual classes and interfaces. In the real example of this chapter, a Blazor application is used as the client application, and the interaction with the database is mocked using

objects in the memory. Following the diagram demonstrated in [Figure 7.1](#), the first class must be created and defined for each factory, starting with the **Text Message Factory** and its underlying interface, with the scope limited to the representation shown in [Figure 7.2](#):

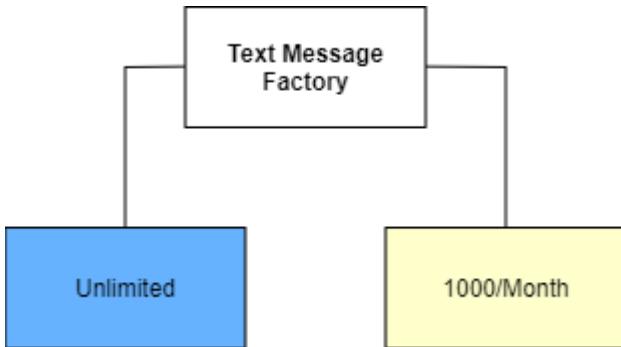


Figure 7.2: Text Message Factory representation

The implementation of the **Text Message Factory** requires an interface that includes properties related to the underlying concrete type for the category (unlimited and 1,000/month), as shown in [Figure 7.3](#):

```
1 namespace Chapter7.Models
2 {
3     0 references
4     public interface ITextMessageFactory
5     {
6         0 references
7         string Name { get; set; }
8         0 references
9         string QualityPerMonth { get; set; }
10    }
```

Figure 7.3: Text Message Factory interface

The interface contains only two properties, which aim to identify the difference between the objects when other classes in the given scenario will consume them. Creating interfaces for the **Abstract Factory** classes is a good practice in terms of future testability and extensibility for any project, and the next step in our example is to create the concrete class that implements the given interface for the

Text Message Factory, as shown in [Figure 7.4](#):

```
1 namespace Chapter7.Models
2 {
3     0 references
4     public class ThousandTextMessage: ITextMessageFactory
5     {
6         1 reference
7         public string Name { get; set; } = "A thousand text messages";
8
9         1 reference
10        public string QuantityPerMonth { get; set; } = "1000";    }
11 }
```

Figure 7.4: Thousand Text Message class

As the given scenario contemplates two types of text message plans (unlimited and a thousand messages), the second class regarding the unlimited type must be created, implementing the same interface **ITextMessageFactory**, as shown in [Figure 7.5](#):

```
1 namespace Chapter7.Models
2 {
3     0 references
4     public class UnlimitedTexxtMessage: ITextMessageFactory
5     {
6         1 reference
7         public string Name { get; set; } = "Unlimited Text Messages";
8
9         1 reference
10        public string QuantityPerMonth { get; set; } = "Unlimited";
11    }
12 }
```

Figure 7.5: Unlimited Text Message class

Considering both types implement the same interface, they have the same members and methods. However, each can implement a distinct behavior based on each specification. This approach allows us to create units and tests consistently, mock integration with databases, inject data and extend the functionality to other types without breaking the existing implementation.

As stated in the requirements for the scenario in this chapter, a mobile plan must contain information on text messages, internet connection, and mobile data plan, as shown in the list of requirements at the beginning of this section. Considering the infrastructure for the text message is already created, the next step is to create the **Internet Connection Factory**, which should follow the structure presented in [Figure 7.6](#):

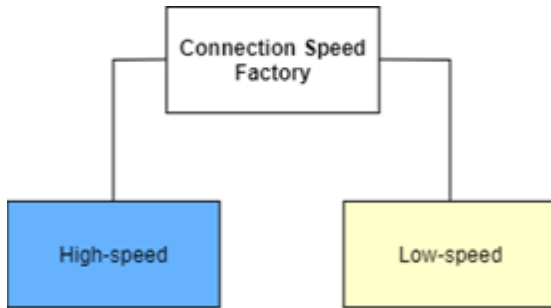


Figure 7.6: Connection Speed Factory representation

The implementation of the **Connect Speed Factory** requires an interface that includes properties related to the concrete type (high speed and low speed), using interfaces as shown in [Figure 7.7](#):

```
1 namespace Chapter7.Models
2 {
3     0 references
4     public interface IConnectionSpeedFactory
5     {
6         0 references
7         string Name { get; set; }
8         0 references
9         string Velocity { get; set; }
10    }
```

Figure 7.7: Connection Speed Factory interface

The interface contains only two properties (**Name** and **Velocity**), which aim to identify the difference between the objects that the implementation class will consume. The next step is to create the concrete class that implements the interface for the Connection Speed Factory, as shown in [Figure 7.8](#):


```

1  namespace Chapter7.Models
2  {
3      0 references
4      public class LowSpeed: IConnectionSpeedFactory
5      {
6          1 reference
7          public string Name { get; set; } = "Low Speed";
8          1 reference
9          public string Velocity { get; set; } = "50 Mb/Sec";
10     }
11 }

```

Figure 7.8: Low-Speed Factory class

As the given scenario contemplates two types of connection speed (high speed and low speed), the second class regarding the high-speed type must be created, implementing the same interface **IConnectionSpeedFactory**, as shown in [Figure 7.9](#):

```

1  namespace Chapter7.Models
2  {
3      0 references
4      public class HighSpeed: IConnectionSpeedFactory
5      {
6          1 reference
7          public string Name { get; set; } = "High Speed";
8          1 reference
9          public string Velocity { get; set; } = "500 Mb/Sec";
10     }
11 }

```

Figure 7.9: High-speed class

Considering both types share the same interface, it is mandatory to implement the members and methods with similar signatures, but each of the classes might implement a distinct behavior based on their specification. The mobile plan requires the creation of an extra and last factory class responsible for handling the creation of the mobile data transfer information, which must have two different types, as shown in [Figure 7.10](#):

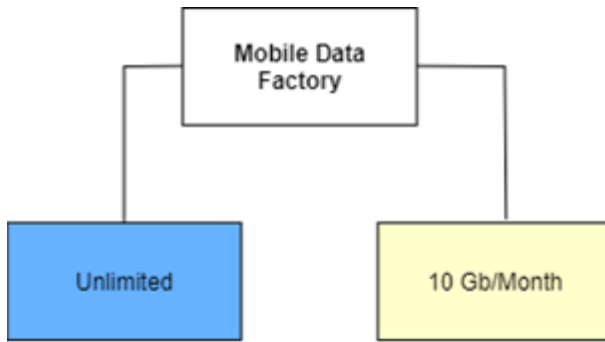


Figure 7.10: Mobile Data Factory representation

The implementation of the **Mobile Data Factory** requires the creation of an interface that includes the properties related to the future concrete types (unlimited and 10 Gb/month), sharing the same interface, as shown in [Figure 7.11](#):

```
1  namespace Chapter7.Models
2  {
3      0 references
4      public interface IMobileDataFactory
5      {
6          0 references
7          string Name { get; set; }
8          0 references
9          string Limite { get; set; }
10     }
```

Figure 7.11: Mobile Data Factory interface

The interface has two main properties (name and limit), which aim to identify the difference between the different types of objects. The next step is to specify the concrete class that implements the interface for the **Mobile Data Factory**, as shown in [Figure 7.12](#):

```

1  namespace Chapter7.Models
2  {
3      1 reference
4      public class UnlimitedMobileData: IMobileDataFactory
5      {
6          1 reference
7          public string Name { get; set; } = "Unlimited";
8          1 reference
9          public string Limit { get; set; } = "Unlimited";
10     }
11 }

```

Figure 7.12: Unlimited Mobile Data class

Considering the given scenario of this chapter requires two types of unlimited mobile data plans (unlimited and 10 Gb), the second class regarding the limit of 10 Gb for the pre-paid option must be created, implementing the same interface **IMobileDataFactory**, as shown in [Figure 7.13](#):

```

1  namespace Chapter7.Models
2  {
3      0 references
4      public class TenGigabytes: IMobileDataFactory
5      {
6          1 reference
7          public string Name { get; set; } = "10 Gb/sec";
8          1 reference
9          public string Limit { get; set; } = "10 Gb/sec";
10     }
11 }

```

Figure 7.13: Ten Gigabytes Mobile Data class

At this stage, all the necessary sub-classes regarding the properties for the main factory are already created, making possible the implementation of the pre-paid and post-paid options, which will be responsible for handling the custom creation of each mobile plan obeying all the requirements determined at the beginning of this section. The Abstract Factory Pattern allows us to create complex objects for abstracting and encapsulating details of their implementation, making the code easier to read and understand for developers. In a real-case scenario, the mobile plan factory must be responsible for creating any different type apart from the two plans, following the business requirements that may have constant changes over time from stakeholders and from the new demands of

the market. Because of that, if new types need to be added, the main class responsible for the creation will hide the complexity of the creation of the new types, and functionality can be extended safely without huge concerns about adding issues to the software architecture.

Similar to what was done with the factories for the sub-classes (text message, mobile data, and connection speed), the next step for the Abstract Factory Pattern implementation is to create the infrastructure responsible for creating the compound object related to the pre-paid and post-paid options for mobile plans represented in [Figure 7.14](#):

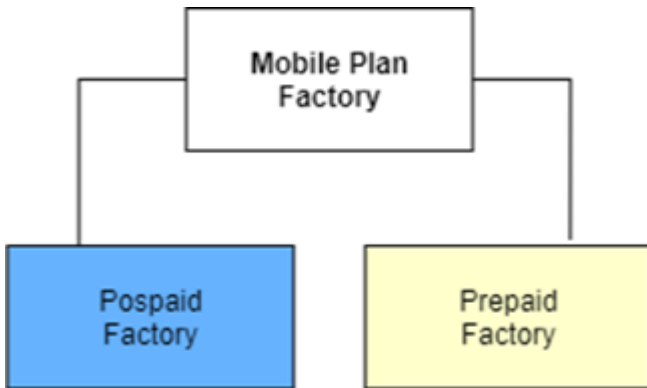


Figure 7.14: Mobile plan factory representation

To facilitate tests and increase reusability, it is always recommended to use interfaces and a main abstract factory class, which must refer to the other interfaces instead of pointing directly to concrete classes the in the definition of its properties, as shown in the following [Figure 7.15](#):

```

1 namespace Chapter7.Models
2 {
3     0 references
4     public interface IMobilePlanFactory
5     {
6         0 references
7         ITextMessageFactory CreateTextMessage();
8         0 references
9         IConnectionSpeedFactory CreateConnectionSpeed();
10        0 references
11        IMobileDataFactory CreateMobileData();
12    }
13 }

```

Figure 7.15: Mobile Plan Factory interface

The **IMobilePlanFactory** interface contains three properties, one for each characteristic determined by the requirements regarding text message, connection speed, and mobile data. As a convention, the Abstract Factory Pattern uses the prefix **Create** related to the generation of each property. It helps identify the pattern by other developers who know how the Abstract Factory Pattern works. Also, it is recommended to use the suffix **Factory** in the name of the classes to correlate the design pattern. The use of meaningful names in software development is one of the most important good coding practices because it allows us to reduce the amount of project documentation and a chance to decrease the time spent analyzing the code written by others.

Once the interface for the mobile plan factory is already created, the next step is to create the two mobile plans available, according to the requirements, following their individual specification strictly. First of all, the **PrepaidFactory** class must be created to implement the underlying interface, as shown in the following

Figure 7.16:

```

1 namespace Chapter7.Models
2 {
3     0 references
4     public class PrepaidFactory: IMobilePlanFactory
5     {
6         1 reference
7         public IConnectionSpeedFactory CreateConnectionSpeed()
8         {
9             return new LowSpeed();
10        }
11
12        1 reference
13        public IMobileDataFactory CreateMobileData()
14        {
15            return new TenGigabytes();
16        }
17
18        1 reference
19        public ITextMessageFactory CreateTextMessage()
20        {
21            return new ThousandTextMessage();
22        }
23    }
24 }

```

Figure 7.16: Pre-paid Mobile Plan Factory class

The **PrepaidFactory** class creates the instances of related properties following their requirements, such as the **LowSpeed** class instance in the method **CreateConnectionSpeed**, and the same for the other two creation methods regarding the other properties, which also refer to the interfaces. On the other hand, the **PostpaidFactory** class automatically populates its properties with different types according to its own specification, as shown in [Figure 7.17](#):

```

1  namespace Chapter7.Models
2  {
3      0 references
4      public class PostpaidFactory: IMobilePlanFactory
5      {
6          1 reference
7          public IConnectionSpeedFactory CreateConnectionSpeed()
8          {
9              return new HighSpeed();
10         }
11
12         1 reference
13         public IMobileDataFactory CreateMobileData()
14         {
15             return new UnlimitedMobileData();
16         }
17
18         1 reference
19         public ITextMessageFactory CreateTextMessage()
20         {
21             return new UnlimitedTexxtMessage();
22         }
23     }
24 }

```

Figure 7.17: *Postpaid Mobile Plan Factory class*

This approach helps to make the code interpretation relatively easy, considering that each factory class explicitly creates its own instances of different objects, even though they share a similar interface. The last step in implementing the given scenario for the mobile plan is creating the client class, the highest level in the hierarchical structure of the Abstract Factory Pattern. This client class does not need to be exposed to details on the deeper levels of the hierarchy because the sub-factory classes handle the complexity and implementation of each object creation by respecting the underlying requirements for the pre-paid and post-paid mobile plans. The client class must implement the logical statement with the necessary rules to create the correct instance of the factory class based on the mobile plan type. To clarify the purpose of the code, it is possible to create an enum in this context, referring to the two types of mobile plans, as shown in [Figure 7.18](#):

```

1  namespace Chapter7.Models
2  {
3      0 references
4      public enum MobilePlan
5      {
6          Prepaid = 1,
7          Postpaid = 2
8      }
9  }

```

Figure 7.18: Mobile plan enum

Furthermore, the client class can receive in the constructor a specific mobile plan type and can also contain a logical statement to create the underlying mobile plan factory, as demonstrated in [Figure 7.19](#):

```

3  public class MobilePlanClient
4  {
5      private IMobilePlanFactory _mobilePlanFactory;
6      0 references
7      public MobilePlanClient(MobilePlan mobilePlan) {
8          if(mobilePlan == MobilePlan.Prepaid)
9          {
10             _mobilePlanFactory = new PrepaidFactory();
11          }
12          else
13          {
14             _mobilePlanFactory = new PostpaidFactory();
15          }
16      }
17
18      0 references
19      public string Describe()
20      {
21          return @"Mobile Plan: {_mobilePlanFactory.CreateConnectionSpeed().Name}.
22                  Text Message: {_mobilePlanFactory.CreateTextMessage().Name}.
23                  Internet Connection: {_mobilePlanFactory.CreateConnectionSpeed().Name}";
24      }
25  }

```

Figure 7.19: Mobile plan enum

In the given scenario, the class's constructor receives a mobile plan type referring to the **enum** previously created, as there is an **if** statement on this method that checks the type. Therefore, based on its value, a distinct instance of a mobile plan factory is created: the **Prepaid** or **Postpaid** factory. Additionally, a method called **Describe** is specified to show the difference between the objects only for demonstration purposes.

In a Blazor Server application, we can create one or multiple

instances of the `MobilePlan` class and show the result of the `Describe` method, as shown in [Figure 7.20](#):

```
1  @page "/"
2  @using Chapter7.Models;
3
4  @if(prePaidPlanClient != null)
5  {
6      <b>Pre-paid plan:</b>
7      <p>
8          @prePaidPlanClient.Describe();
9      </p>
10 }
11
12 @if (postPaidPlanClient != null)
13 {
14     <b>Post-paid plan:</b>
15     <p>
16         @postPaidPlanClient.Describe();
17     </p>
18 }
19
20 @code{
21     MobilePlanClient prePaidPlanClient =
22     new MobilePlanClient(MobilePlan.Prepaid);
23
24     MobilePlanClient postPaidPlanClient = new
25     MobilePlanClient(MobilePlan.Postpaid);
26
27 }
```

Figure 7.20: Blazor server implementation

As shown in [Figure 7.20](#), the `Index.razor` component in a Blazor Server application creates two instances of the `MobilePlanClient` class, each of them with a different type (`Prepaid` and `Postpaid`), and shows on the Web page the description of each mobile plan. After running the Blazor Server application, you will be able to see the specification for each type of mobile plan on the screen, as shown in [Figure 7.21](#):

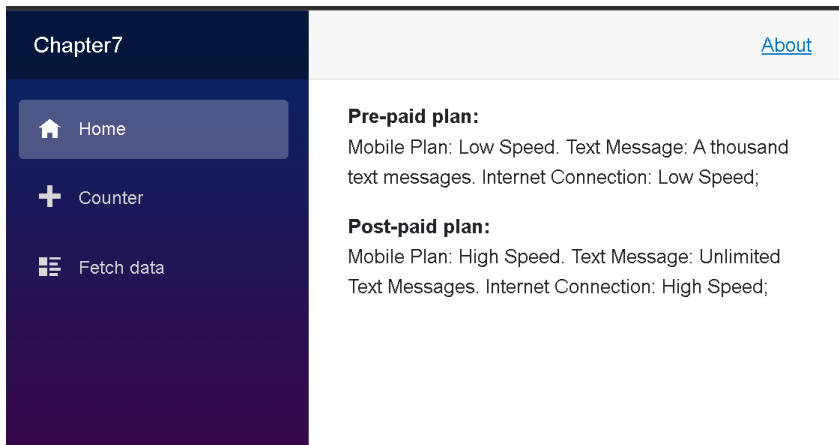


Figure 7.21: Blazor server result

The use of the Abstract Factory classes by the Mobile Client class is relatively simple because it encapsulates the implementation of each mobile plan type from the highest level of the application, delegating all the responsibility and details on the implementation to the specialized classes.

Conclusion

As you learned in this chapter, the Abstract Factory Pattern helps manage the creation of families of objects, keeping an elegant and sophisticated software architecture that allows us to extend, test, and maintain the related functionality without compromising the readability aspects of the code. Considering that the pattern extensively uses interfaces, it is possible to have a consistent structure and take all the benefits of object-oriented programming practices by following the highest recommended approaches in scenarios similar to the one demonstrated in this chapter.

In this chapter, you have learned how to implement the Abstract Factory Pattern in a real scenario using C# language, and you became able to understand the central points of design patterns in general, which is the possibility of using general solutions for common problems in software development in terms of implementation and software architecture.

In the next chapter, you will continue your journey into Design Patterns using C# language and the .NET platform, learning the

Prototype pattern, and applying its concept in an ASP.NET Core Razor application.

Points to remember

- The Abstract Factory is conventionally considered as the factory of other factories.
- The primary purpose of the Abstract Factory Pattern is to build complex objects without exposing their implementation.
- The C# language fully supports the implementation of the Abstract Factory method once it is based on interfaces and abstract classes, which are standard types in the object-oriented programming paradigm.

Multiple choice questions

1. **Which is the recommended suffix to be used in a class name using an abstract factory pattern?**
 - A. Interface
 - B. Factory
 - C. Abstract
 - D. Create
2. **What definition better represents the abstract factory pattern?**
 - A. It is a design pattern that allows us to create a single instance of an object for the entire application.
 - B. The pattern represents a way to create multiple objects of the same type.
 - C. This pattern allows us to create multiple object families by encapsulating the implementation of each family of objects in other sub-factories.
3. **Which alternative contains some benefits of the abstract factory pattern?**
 - A. Testability, extensibility, and easy maintenance
 - B. An abstraction of database operations

C. Performance and scalability

Answer

	3	
	2	
	3	

Questions

1. Explain the main purpose of the Abstract Factory pattern.
2. Why is the use of interfaces necessary in the Abstract Factory pattern?
3. According to what you have learned in this chapter, create a hypothetical scenario where the abstract factory could be applied, apart from the example given in this chapter.

Key terms

- **Abstract Factory:** This is the main interface or abstract class that declares the creation methods for a set of products that are part of the family. It typically contains a method for each type of product to be created.
- **Concrete Factory:** These classes or objects implement the methods defined in the Abstract Factory. Each Concrete Factory corresponds to a specific variant/family of products and creates those product objects.
- **Abstract Product:** This represents the main interface or abstract class for a type of product. It declares the operations that the product must implement.
- **Concrete Product:** These are specific implementations of the Abstract Product. They are created by the corresponding Concrete Factory and represent a specific variant of the product.
- **Client Code:** This is the part of the application that uses the Abstract Factory and its products. The client code is decoupled from the actual concrete classes, relying only on the abstract interfaces. This allows the client to work with different

families of products without changing its core code.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Prototype Pattern with ASP.NET Razor

Introduction

Many cases in software development require the creation of new instances of complex objects repeated times, which may cause performance issues and extra challenges in terms of the usage of computational resources, including memory consumption and CPU, representing a limitation for scaling modern applications. The Prototype Pattern allows us to create copies of complex and costly objects, primarily used in .NET applications and C# libraries.

Learning the Prototype Pattern gives you a chance to understand which scenarios this pattern can be a handful of and also gives you enough knowledge to recognize the pattern in legacy projects based on C# language.

Structure

In this chapter, we will discuss the following topics:

- Prototype pattern concept

- Examples in C# and .NET
- Details on razor pages project type

Objectives

After studying this unit, you should understand which scenarios the prototype pattern can be applied, identify the use of the prototype pattern in legacy projects and third-party libraries, and apply the prototype pattern in real projects.

Prototype pattern definition

The prototype pattern primarily allows the efficient copy of complex objects, saving resources in creating new instances of objects that require a substantial computational effort to be generated. With the use of the prototype pattern, the whole hard part of the creation of complex objects is essentially hidden from the consumer once the high cost of the process occurs, ideally just once, and any other extra instances of the same objects are made using an approach that allows the creation of a full copy of a template of the object structure, modifying only specific properties to generate the correct state. Therefore, the first time the object is created, a model for new instances is generated, keeping the correct initial state for most properties, with a specialized state for particular properties for any new instances.

The prototype pattern is part of the creational patterns once it helps in the generation of objects, and in specific scenarios, it is used in combination with other creational patterns, such as abstract factory and builder, to give an excellent performance to these other patterns.

Any factory pattern has the responsibility of simplifying the creation of complex structures to be justified, but the prototype pattern is much more straightforward than the others, considering it does not require a complex set of classes to work correctly, and the prototype pattern does not need the use of inheritance, and multiple interfaces to achieve its purpose. However, all the complexity behind creating the first complex object to be used as a template for others requires additional implementation logic. Each scenario states a custom analysis on which approach would be better, always thinking in

terms of simplicity, maintenance, extensibility, and other development costs.

In software developed using the Object-Oriented Programming paradigm, sometimes, it is not allowed to copy specific properties of objects because they are marked with the private modifier, or they might require extra logic to be correctly populated. In that case, the prototype pattern represents an excellent option, as it facilitates the copy of all the properties of objects, keeping the desired state for those. This approach mainly uses packages and libraries to manage the copy of complex structures, such as large XML objects, and ORM database contexts, among many other scenarios.

Given the brief explanation of the theoretical concepts and purpose of the prototype pattern, the next section will show you how to apply it in a real scenario with a practical example, and it will allow you to see how the prototype pattern can be used natively in distinct .NET projects with C# language.

Prototype pattern implementation

Considering the prototype pattern is closely related to the strategy of the copy of objects, before starting to apply the prototype pattern in a real example, you must understand the difference between a shallow copy and a deep copy in C# language, which will help you to make the right decisions in terms of software architecture and will allow you to prevent unexpected software behavior, mainly in production environments.

Usually, in scenarios where it is necessary to copy the state of an object to another, both of the objects involved share the address in the memory depending on how the copy of the object is done. Therefore, if a particular routine in the software changes one of the instances of the object that is copied, all the other copies will be affected in terms of state and their properties changed at the same time. Although they apparently are different instances, they share the same reference in the memory and are physically stored in the same place. That type of copy that shares the same address in the memory is usually called **Shallow Copying** in C# language, and it can be achieved using the equal operator for assignment.

To check how the Shallow Copy works in practice, create a new

ASP.NET Core Web App using Visual Studio 2022, selecting the template highlighted in *Figure 8.1*:

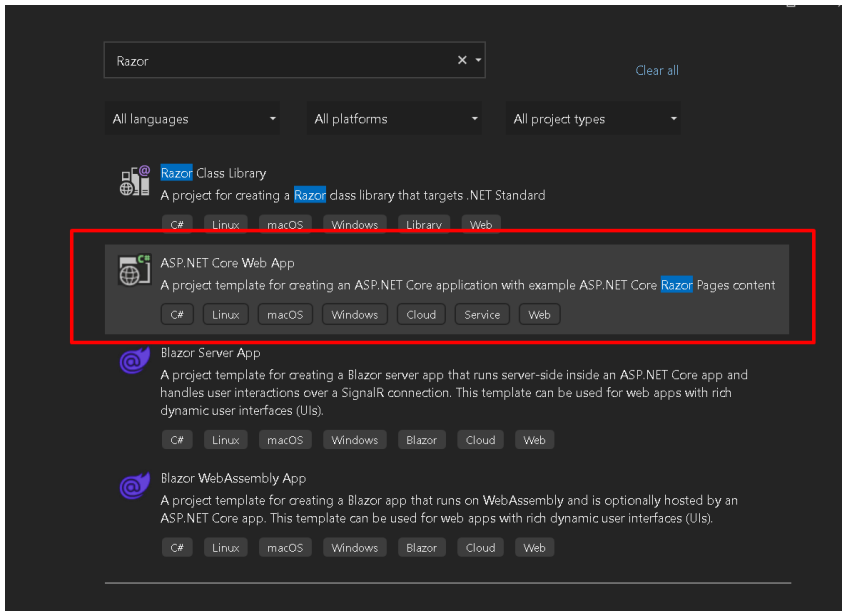


Figure 8.1: Razor Pages project

The ASP.NET Core Web App template creates a set of default folders and files, following a standard structure for a Razor Pages project, which does not have the traditional Controllers from the MVC architecture, as seen in *Figure 8.2*:

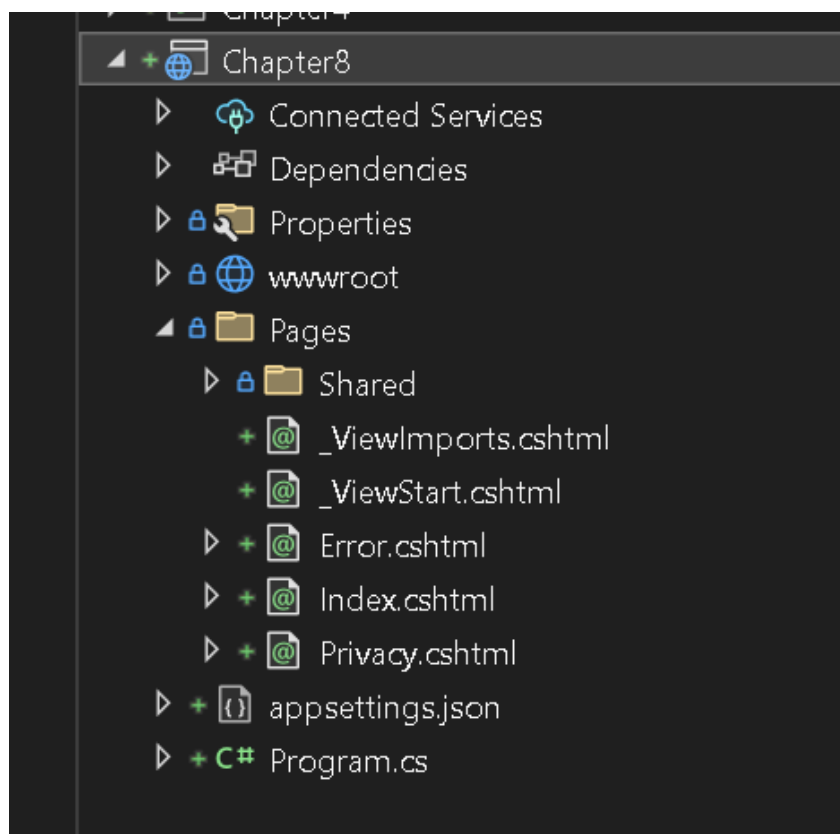


Figure 8.2: *Razor Pages Project structure*

Once the project is generated, create a class called **Customer** within the project, which will be used to demonstrate the Shallow Copy. The structure of the class can follow the example in [Figure 8.3](#):

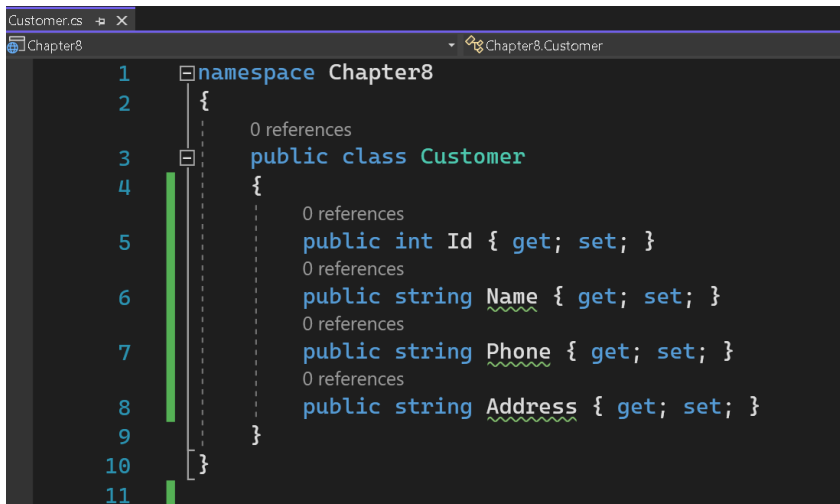


Figure 8.3: Customer class

Razor Pages projects work with the concept of Models to render information in the application's front end. To make the test of the Shallow Copy feature in C# language, modify the `Index.cshtml.cs` file your project to include a property related to a List of Customers and change the `OnGet` method body to create two `Customer` objects, with the second one being a copy of the first object, as seen in [Figure 8.4](#):

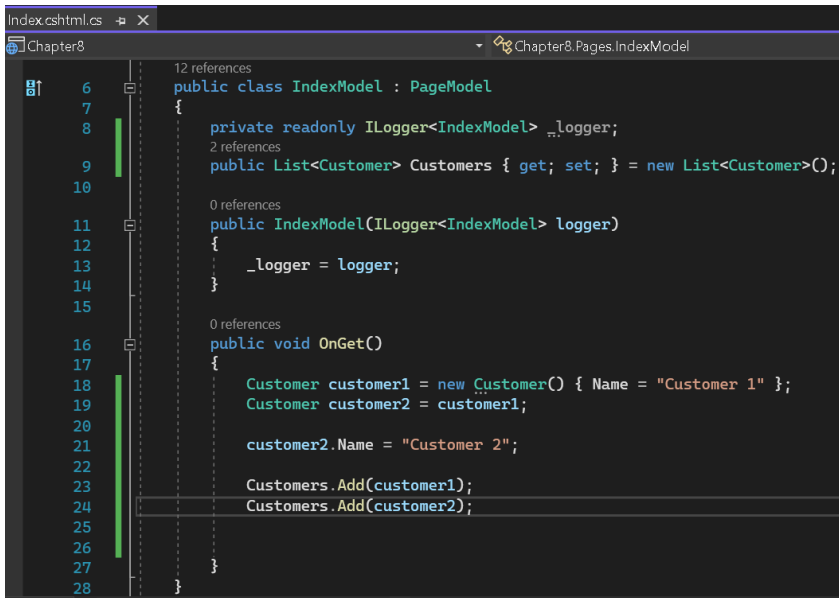


Figure 8.4: Backend code for the Index Razor page

As you can see in line 19, the **customer2** variable is a copy of the **customer1** variable, and line 21 changes the **Name** property to **Customer 2**". After this, the two objects are included in the **Customers** list. In order to see the results of the state of these two objects, make the necessary modifications to the **Index.cshtml** file, to loop through the **Customers** List and display the name property as seen in [Figure 8.5](#):

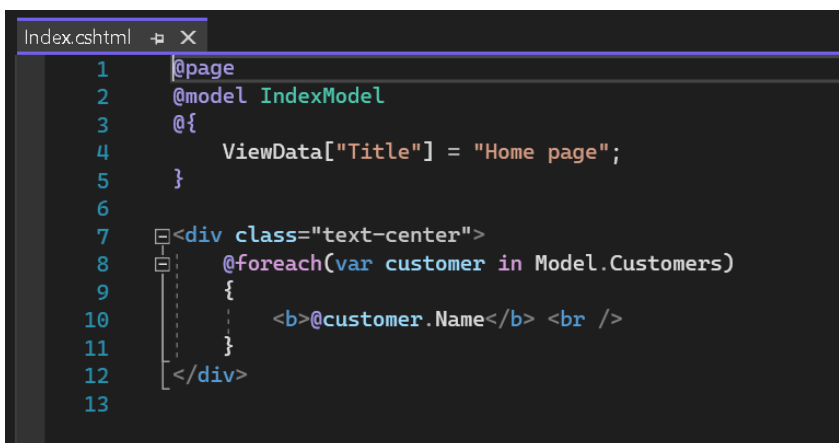


Figure 8.5: Index Razor page

Considering the second customer object is being created as a copy from the first instance using the equal operator, all the changes made in any instances automatically reflect into the other instances. If we run the application, the Razor page, it is possible to realize after the Index page renders that the two customers have the same name, as seen in [Figure 8.6](#):

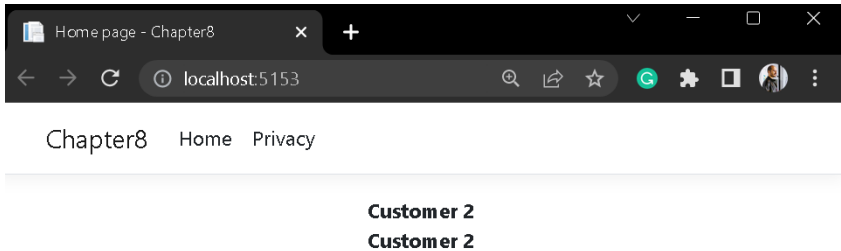


Figure 8.6: Shallow copy result

Both objects share the same address in the memory; therefore, programmatically speaking, they are distinct objects. This behavior may cause a few issues for developers unfamiliar with C# language, and it is not hard to introduce bugs in projects of any kind if the correct approach for copies of objects is not used.

Additionally, there is another way to make shallow copies, which consists of implementing the `ICloneable` interface. This interface forces the implementation of the `Clone` method, being possible to use a native method called `MemberwiseClone` to apply the shallow copy. The clone method keeps the same reference in the memory for reference-type properties. To test this behavior, you can implement the code shown in [Figure 8.7](#), which changes the `Customer` class:

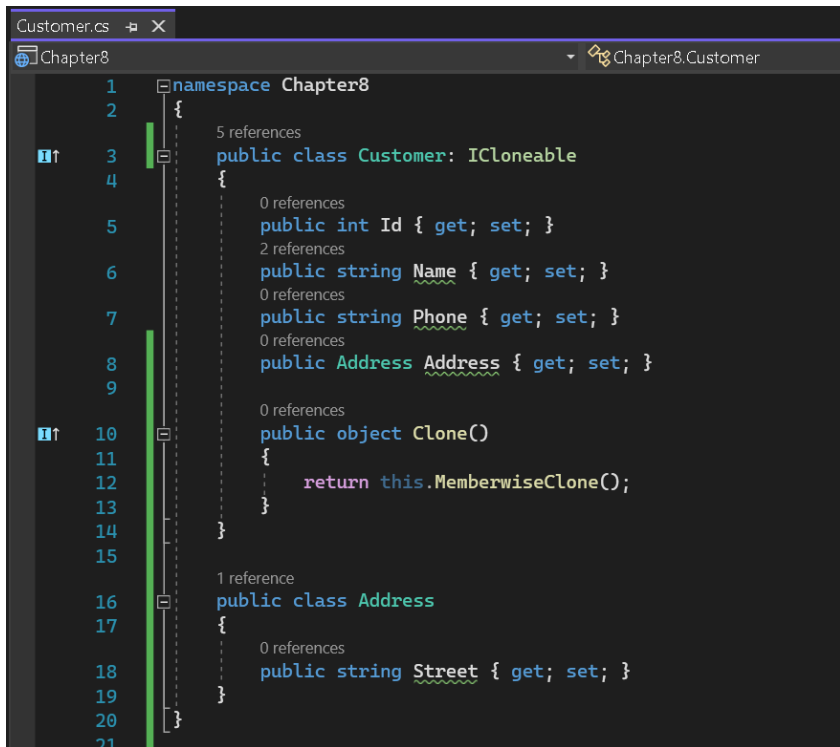
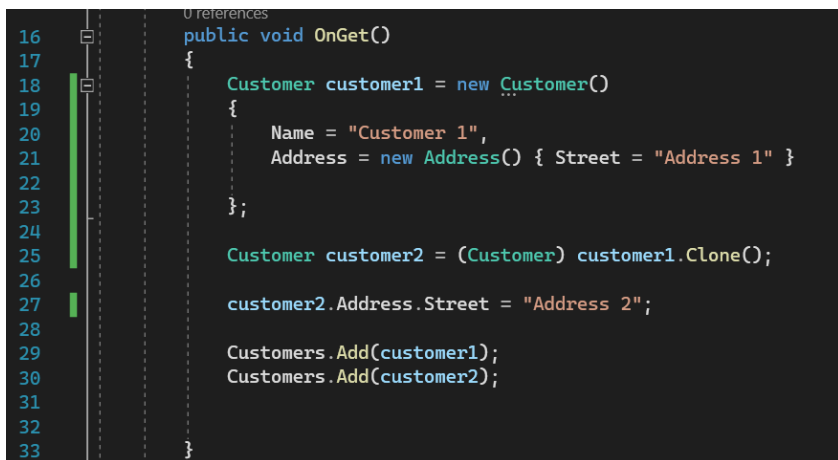


Figure 8.7: MemberwiseClone method

Note that the **Customer** class implements the **ICloneable** interface specified on its signature. An **Address** class is specified to change the underlying property of the **Customer** class to a non-primitive type, which will be helpful to test the behavior of Shallow copies for non-primitive types as well.

With the implementation of the **ICloneable** interface, the **clone** method can be used in the **onGet** method in the backend of the Index Razor page, as shown in [Figure 8.8](#):



```

0 references
public void OnGet()
{
    Customer customer1 = new Customer()
    {
        Name = "Customer 1",
        Address = new Address() { Street = "Address 1" }
    };

    Customer customer2 = (Customer) customer1.Clone();

    customer2.Address.Street = "Address 2";

    Customers.Add(customer1);
    Customers.Add(customer2);
}

```

Figure 8.8: Clone method

In this case, the property **Address** is not a primitive type in C# language. When a copy is created using the standard **Clone** method, all the properties can be normally changed in the copy. Still, the **Address** property shares the same memory location as the source object in the copy from the **customer1** object.

As an alternative to the Shallow Copy option explained in this chapter, the C# language allows us to use the concept of **Deep Copy**, which consists of the creation of actual independent copies of objects that do not share the same reference in the memory, and a change in any of the objects involved in the copy do not affect the other instances. In the majority of cases in software development, this is the expected behavior of copies of objects. However, to avoid performance issues in the deep copy of complex objects, it is imperative to understand and learn how to create deep copies correctly. After changing the **Clone** method of the previous example, the source object needs to be serialized in a memory stream object, and the return of the method must be the deserialization of the same object, as shown in [Figure 8.9](#):

```

12 |         | reference
13 |         | public object Clone()
14 |         | {
15 |         |     using(var memoryStream = new MemoryStream())
16 |         |     {
17 |         |         var binaryFormatter = new BinaryFormatter();
18 |         |         binaryFormatter.Serialize(memoryStream, this);
19 |         |         memoryStream.Position = 0;
20 |         |
21 |         |         return (Customer)binaryFormatter.Deserialize(memoryStream);
22 |         |     }
23 |         | }

```

Figure 8.9: Deep Copy

This new version of the `Clone` method guarantees that a new reference in memory is created for the copy. Therefore, any changes made in the source object do not affect any copies because they are completely independent from each other. The code presented in [Figure 8.9](#) demonstrates how to make deep copies of objects without requiring to specify property by property of the objects individually.

Considering that you already learned the difference between deep and shallow copy in C# language, it is time to get familiar with the prototype pattern by following a practical example. For this, imagine a hypothetical scenario where you are working on a project related to a streaming platform which gives the customers opportunity to subscribe to monthly plans. Each plan has its specifications, and they technically represent a complex class in C# language whose instances must be populated in runtime, applying business requirements. If the company that offers the streaming service decides to upgrade the policies for a specific plan, only the version of the plan and its price needs to be changed, but the rest of the properties remain the same.

In this context, it would be an advantage to create a copy of the previous version of the plan and change only the relevant properties in the copy. It would allow us to do an efficient use of the memory in the application server and it would potentially increase the performance for users, mainly if that copy represents a complex operation in terms of computational resources. Each design pattern has the purpose of solving a particular problem in software development, and there must be always a good reason to apply some of them in real scenarios. In the context of the scenario given in this chapter, the requirements clearly state that the customer plan in the streaming service that needs to be upgraded is still the same, exception for the version number and price. Depending on

the complexity of the creation of new plans in the application, the user of the prototype pattern is fairly applicable, considering it has the purpose of making efficient copies of objects.

The given scenario must have two classes that represent the plan for customers, as shown in [Figure 8.10](#):

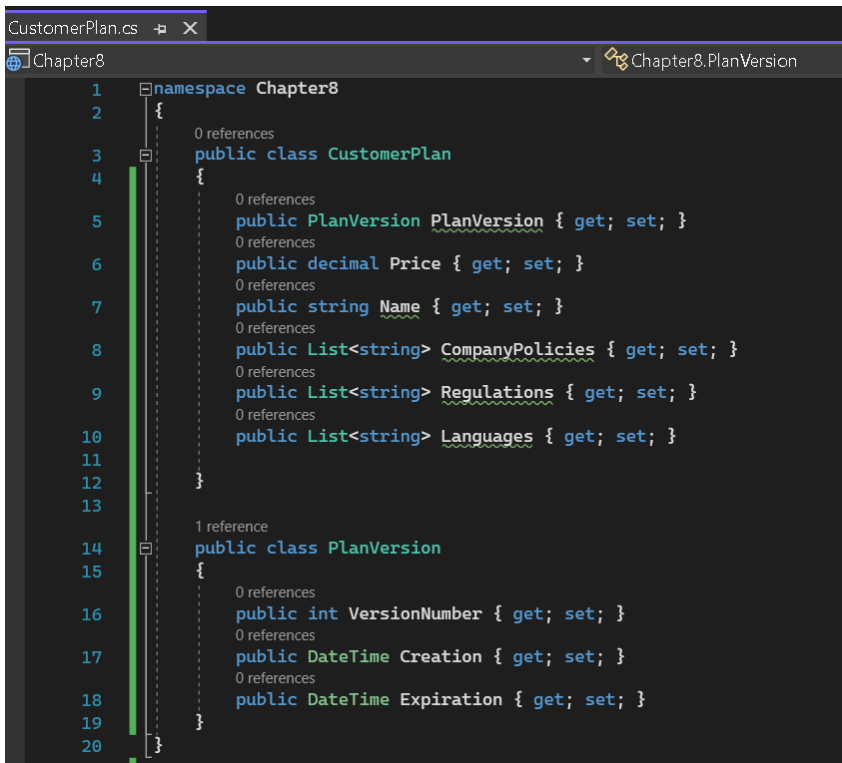


Figure 8.10: Customer Plan class

As shown in [Figure 8.10](#), the **CustomerPlan** class contains properties regarding company policies, regulations, and extra information. According to the requirements, these properties regarding policies and regulations must have the same values for all customer plans, but the price and version can vary between them. Considering the explanation on shallow copy given in this chapter, if a developer does not implement the copies correctly in the application, the lack of knowledge on copies of referenced types in C# language may cause issues to achieve the requirements in the scenario, with multiple instances of customer plans being updated

by mistake, as you can see in [Figure 8.11](#):

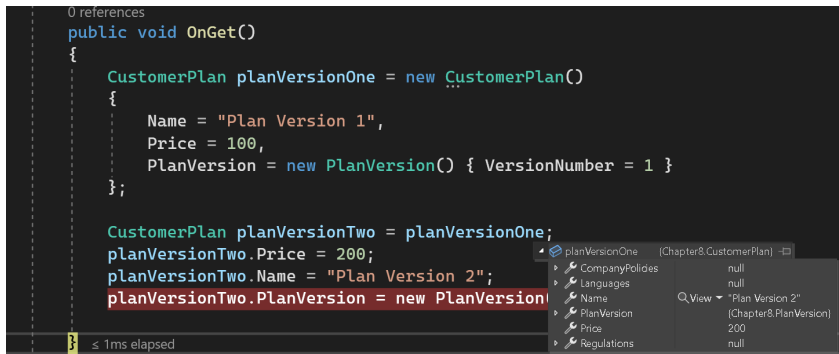


Figure 8.11: Second shallow copy example

The code present in [Figure 8.11](#) creates two distinct instances of the **CustomerPlan** class assigning different values for the name, price and version properties. What is the issue with this code?

Considering there is a requirement to keep the same value only for the company policies and regulations properties, the routine copies the first plan to create the second customer plan using the equal operator. When that is done, the rest of the code changes the values that should be applicable only for the second plan (name, price and version), but as you can see in the details on Debug mode for the first object, you can realize that the state of the first customer plan is the same as the second plan.

In this case, both of the customer plans have the same name, price and version, because the two objects share the same memory reference. The result is clearly not matching the business requirements. For solving this problem, while keeping an excellent performance for the routine, it is possible to apply the prototype pattern using the **Deep Copy** approach shown previously in this chapter. There is the option of using the **ICloneable** interface as well; however, in this chapter a more explicitly **Deep Copy** method will be implemented, to make the implementation of the requirements clear in the code, as seen in [Figure 8.12](#):

```

5 public class CustomerPlan
6 {
7     2 references
8     public PlanVersion PlanVersion { get; set; }
9     2 references
10    public decimal Price { get; set; }
11    2 references
12    public string Name { get; set; }
13    0 references
14    public List<string> CompanyPolicies { get; set; }
15    0 references
16    public List<string> Regulations { get; set; }
17    0 references
18    public List<string> Languages { get; set; }
19
20    0 references
21    public object DeepCopy()
22    {
23        using(var memoryStream = new MemoryStream())
24        {
25            var binaryFormatter = new BinaryFormatter();
26            binaryFormatter.Serialize(memoryStream, this);
27            memoryStream.Position = 0;
28
29            return (CustomerPlan) binaryFormatter.Deserialize(memoryStream);
30        }
31    }
32 }

```

Figure 8.12: Deep Copy method

The properties for the **CustomerPlan** class remain the same, but a new method called **DeepCopy** is now part of the class structure, which allows us to create actual copies of objects without sharing the same memory address. Considering the **DeepCopy** method is using serialization, all the classes involved in the copy must be marked with the proper **Serializable** attribute, as shown in [Figure 8.13](#):

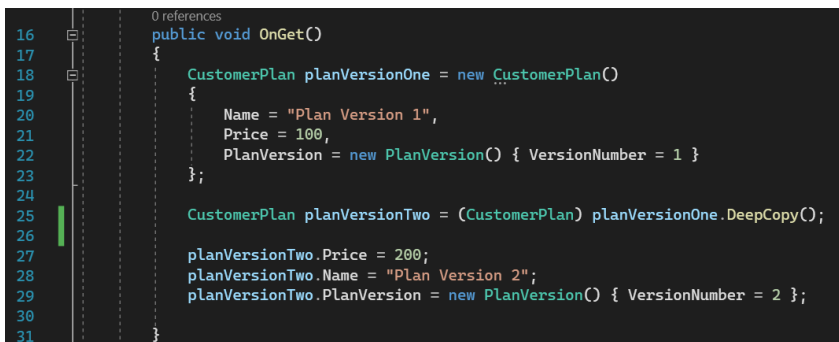
```

CustomerPlan.cs
Chapter8
1 using System.Runtime.Serialization.Formatters.Binary;
2
3 namespace Chapter8
4 {
5     [Serializable]
6     4 references
7     public class CustomerPlan ...
8
9     [Serializable]
10    3 references
11    public class PlanVersion ...
12 }

```

Figure 8.13: Serializable Attribute

With the given modifications, the classes are now totally ready to be used without any runtime exceptions for serialization. Considering the previous code sample in the `OnGet` method, it is necessary to change the creation of the second customer plan, which has to use now the new `DeepCopy` method just created. You can change only the line related to the creation of the second customer plan instance. In that case, the system does not keep the same reference in the memory anymore, and any changes made in one of the plans do not affect any other potential copies. With the required changes in the code, the final version that meets the requirements is presented in [Figure 8.14](#):



```
16 public void OnGet()
17 {
18     CustomerPlan planVersionOne = new CustomerPlan()
19     {
20         Name = "Plan Version 1",
21         Price = 100,
22         PlanVersion = new PlanVersion() { VersionNumber = 1 }
23     };
24
25     CustomerPlan planVersionTwo = (CustomerPlan) planVersionOne.DeepCopy();
26
27     planVersionTwo.Price = 200;
28     planVersionTwo.Name = "Plan Version 2";
29     planVersionTwo.PlanVersion = new PlanVersion() { VersionNumber = 2 };
30
31 }
```

Figure 8.14: Final version of `OnGet` method

As you can see in line 25, the second instance of the customer plan is created as a copy of the first one, but it is using the deep copy method, which guarantees that the two objects will not share the same memory address, and the changes can be safely made in both objects, independently. In a more complex scenario where multiple objects would be created, the gain in performance is much more significant once copies of complex objects have much less costs than the creation of the new instances from scratch.

The prototype pattern allows us to make copies of objects, applying a better approach by copying the properties of another object as a template. It is essential to understand the risk in production environments when memory resources may be over-used, which means that any performance improvement in the code, such as the use of the prototype pattern is very welcome.

Conclusion

As you learned in this chapter, the prototype pattern is helpful in all the scenarios that require the creation of multiple instances of a certain object with similar state, but with distinct values in some of their properties. The prototype pattern is largely used in .NET applications, and it represents an important approach to have a better manage of memory in applications in general and provide to users a good experience in terms of performance.

With this chapter, you have learned how to implement the prototype pattern in a real scenario using the C# language and became able to understand in which situations the pattern can be used. Furthermore, you have learned the difference between shallow and deep copies and understood in which scenario each one is more recommended.

In the next chapter, you will have the chance to continue your journey into design patterns using C# language and .NET platform learning the Factory Method Pattern combined with new features introduced in C# 11.

Points to remember

- The prototype pattern allows us to efficiently create copies of objects, save memory resources, and increase the performance of the software in complex operations that require the creation of multiple similar objects.
- The shallow copy keeps the same reference in the memory for all related object instances.
- The deep copy gives us the possibility to create copies of an object keeping distinct and independent references in the memory.

Multiple choice questions

1. Which interface can be used in the C# language to force the implementation of a clone method?
 - A. IEnumerable
 - B. Interface

- C. ICopy
D. ICloneable
2. Which method can be used to achieve the shallow copy behavior natively in the C# language?
- A. MemberwiseClone
B. ObjectClone
C. Copy
D. Clone

Answer

	D	
	2	

Questions

1. Explain the main purpose of the prototype pattern.
2. Why is the use of prototype pattern important?
3. According to what you have learned in this chapter, create a hypothetical scenario where the prototype pattern could be applied apart from the example given during this chapter.
4. What are the differences between a shallow copy and deep copy in the C# language?

Key terms

- **Shallow copy:** In the C# language, the shallow copy process maintains in the memory the same address reference to all the objects created using a copy method. Therefore, a change in one object affects all the other copies.
- **Deep copy:** In the C# language, the deep copy represents the creation of an object from another one, keeping both of objects in different references in the memory, being the objects totally independent on each other in terms of state changes.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Factory Method Pattern Using New Features on C# 11

Introduction

The correct use of design patterns in software development is crucial in keeping the architecture extensible while simplicity is still being applied to achieve high standards in terms of good practices, maintainability, and flexibility for software projects.

Considering the vast list of available design patterns, the concept implemented on each may sound familiar. In some cases, a quick look at them may result in the inability to observe essential details and understand the difference between them. This may be the case between the Abstract Factory pattern, already explained in [Chapter 7](#), *Abstract Factory Pattern with Blazor*, of this book, and the factory method pattern.

Learning the Abstract Factory pattern gives the programmer a chance to build a simpler architecture and allows the creation of complex objects while applying good practices of the object-

oriented programming paradigm, including polymorphism and encapsulation, developing an alternative to the Abstract Factory structure. This may be prevalent in all scenarios where multiple distinct objects must be created in software without having important subclasses to achieve the same goal.

Structure

In this chapter, we will discuss the following topics:

- Factory Method concept
- Examples in C# and .NET
- Code samples in new features introduced to C# 11

Objectives

After studying this chapter, you should understand the Factory Method pattern for real scenarios and be able to identify the use of the pattern in legacy projects.

Factory Method definition

The Factory Method pattern represents a simpler alternative to the Abstract Factory pattern, allowing us to use the concept of a factory to create complex objects in the case where it is not necessary to generate many subclasses or a relevant number of extra families of classes related to the main object in the highest level of the hierarchy structure responsible for acting as a client service or class.

The Abstract Factory pattern exemplified in [Chapter 7](#), *Abstract Factory Pattern with Blazor*, of this book contains a main factory that creates other factories, which is significantly more complex than the Factory Method pattern. Therefore, its use is recommended when the software architecture is proportionally more complex. It is possible to say that the Factory Method pattern has a similar definition in terms of purposes, but the difference is that this pattern is responsible for creating a factory for objects, and the Abstract Factory pattern is responsible for creating a factory of factories at multiple levels.

The Factory Method defines a base class responsible for creating a

complex object but delegates to the subclasses the responsibility of creating the new instances of the object the application will consume. For example, imagine a scenario where the application needs to register products and services for an online store. The Factor Method might be recommended if the product type is known at the beginning of the implementation of the sales routine and if the objects, in general, do not belong to the same family of projects with multiple subtypes. Furthermore, the creation process of the complex object instance should not involve cascade factories or anything unusual in the implementation of its members.

It is correct to say that this pattern allows us to create complex objects that encapsulate the complexity of the creation of those from the part of the software that primarily consumes the objects. In summary, in the cases where the system needs to create objects that do not contain complex requirements in terms of the state of their properties, the Factory Method represents a good alternative because it hides the implementation and logic behind the factory from the main concrete class that is consuming the objects, allowing us to extend the software architecture by inserting new types of objects over time without compromising consistency.

The correct use of interfaces and abstract classes in any software based on an object-oriented programming paradigm is one of the essential characteristics of having testable, extensible, and reliable software, which represents one of the key points of the Factory Method pattern. Therefore, if you are not familiar with those concepts, it is recommended to review the first chapters of his book, which contain a basic overview of interfaces and good practices regarding object-oriented programming.

Factory Method implementation

As the Factory Method pattern consists of a simpler alternative to the Abstract Factory pattern, it is crucial to apply its concepts in implementing a project with requirements similar to those previously explained in [Chapter 7, Abstract Factory Pattern with Blazor](#), refactoring the architecture to comply with this pattern. Therefore, in this section, you will have the opportunity to implement the base structure of a scenario that involves a company that sells mobile plans to customers with different configurations for

each plan in terms of text messages, mobile data, and other services but with small differences in terms of requirements if compared to the scenario given in [Chapter 7, Abstract Factory Pattern with Blazor](#):

- The company will offer two types of plans to customers: pre-paid and post-paid.
- Both plans must have different conditions for text messages, internet connection speed, and mobile data limits. Only the number of messages and the connection speed are changeable.
- The pre-paid plan has a limited plan for text messages, up to 2,000 messages per month.
- The post-paid option has a limited plan for text messages, up to 5,000 messages per month.
- The pre-paid plan has a maximum connection speed of 50 megabytes per second.
- The post-paid plan has a maximum connection speed of up to 100 megabytes per second.

The requirements clearly show that the mobile plans have the same structure in terms of properties, but the value of the properties for each requirement is different, as shown in [Table 9.1](#):

	Text messages limit	
Pre-paid plan	2,000	
Post-paid plan	5,000	

Table 9.1: Mobile plans

Given that scenario, the use of the Factory Method pattern allows us to build the underlying objects abstracting their implementation from the highest level in the client application, as demonstrated in the diagram in [Figure 9.1](#):

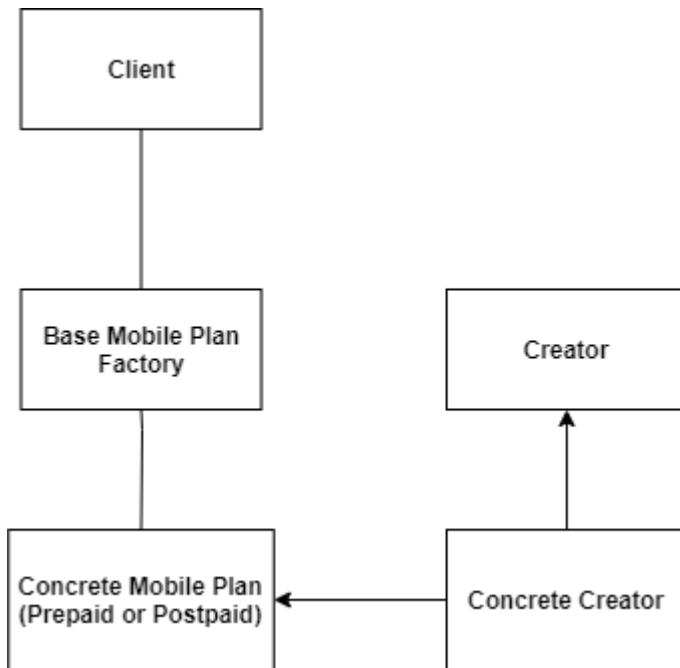


Figure 9.1: Factory method diagram

As you can see in [Figure 9.1](#), the structure of the Factor Method pattern is much simpler than the Abstract Factory pattern, considering that the creation of the sub-factories is not mandatory. Once the requirements are established in this section and the model structure is defined with the proper correlations, we can start the implementation process. The first step is to create individual classes and interfaces. For the example in this chapter, a client application architected as a .NET Console application is sued. For this application, the interaction with the database is not present to keep the focus on the C# implementation for the Abstract Method pattern.

Following the specification in [Figure 9.1](#), we must create the first classes and interfaces to represent the base factory and related classes. We can start by creating the **TextMessage** class, including the underlying interface, with the scope restricted to the representation shown in [Figure 9.2](#):

```

1 namespace Chapter9.Models
2 {
3     1 reference
4     public interface IMessage
5     {
6         1 reference
7         string? Name { get; set; }
8         1 reference
9         int QuantityPerMonth { get; set; }
10    }
11 }

```

Figure 9.2: IMessage interface

The interface contains only two properties, which aim to identify the difference between the objects when other classes in the given scenario will consume them. To begin the coding process for the Factory Method pattern itself, the related interfaces must be created in the very beginning. This is a good practice for keeping a good testability and extensibility level for projects in general.

The next step is to create the concrete class that implements the **IMessage** interface, as shown in [Figure 9.3](#):

```

1 namespace Chapter9.Models
2 {
3     0 references
4     public class TextMessage : IMessage
5     {
6         1 reference
7         public string? Name { get; set; }
8         1 reference
9         public int QuantityPerMonth { get; set; }
10    }
11 }

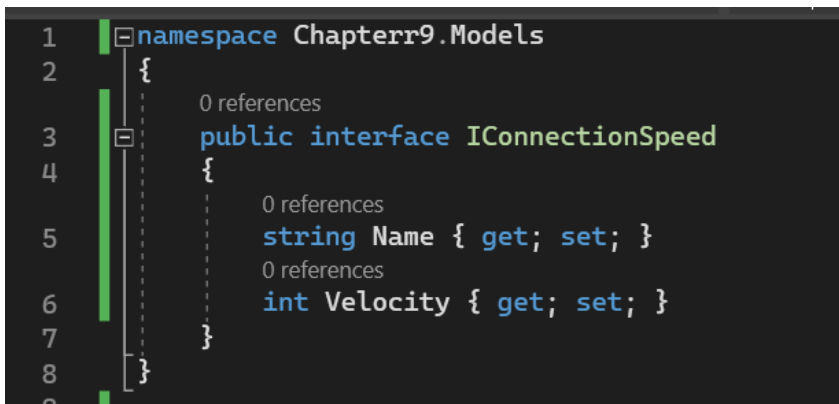
```

Figure 9.3: TextMessage class

The implementation of the **TextMessage** class does not contain any specification regarding the business requirements stated in the scenario, such as a specific value for the **Name** property or a value for the number of text messages allowed in the plan. The full implementation of the requirements should be the responsibility of the factory class instead, which will abstract the logic behind the

text message creation.

A **MobilePlan** in the given scenario must contain information on **TextMessages**, **InternetConnection**, and **MobileDataPlan**, as shown in the list of requirements at the beginning of this section. Considering that the infrastructure for the text message is already created, the next step is to create the **InternetConnection** class, including the related interface. The implementation of the **ConnectionSpeed** structure requires an interface (**IConnectionSpeed**) that includes properties related to the future concrete class, as shown in [Figure 9.4](#):

A screenshot of a code editor showing the definition of the IConnectionSpeed interface. The code is written in C# and is part of a namespace named Chapter9.Models. The interface is public and contains two properties: Name (a string) and Velocity (an int). Both properties have get and set accessors. The code is as follows:

```
1 namespace Chapter9.Models
2 {
3     0 references
4     public interface IConnectionSpeed
5     {
6         0 references
7         string Name { get; set; }
8         0 references
9         int Velocity { get; set; }
10    }
```

Figure 9.4: IConnectionSpeed interface

The interface shown in [Figure 9.4](#) contains only two properties (**name** and **velocity**), which aim to identify the difference between the objects when the implementation class consumes them. The next step is to create the **ConnectionSpeed** concrete class that implements the **IConnectionSpeed** interface, as shown in [Figure 9.5](#):

```

1  namespace Chapterr9.Models
2  {
3      0 references
4      public class ConnectionSpeed : IConnectionSpeed
5      {
6          1 reference
7          public string Name { get; set; }
8          1 reference
9          public int Velocity { get; set; }
10     }
11 }

```

Figure 9.5: ConnectionSpeed class

In the requirements, there are two types of connection speeds (high and low). For this reason, the factory class will be responsible for populating all the properties accordingly for connection speed and other characteristics for each mobile plan. This approach represents a significant decision if the Abstract Factory pattern is used where a concrete class for the distinct sub-types is needed. There are pros and cons to each approach. The best decision must consider the requirements of the project.

The compound properties of the mobile plan class have been created at this point, and the next step is the creation of the mobile plan structure that involves the specification of a common interface called **IMobilePlan**, which will be implemented for the two known mobile plan types (post-paid and pre-paid), as shown in [Figure 9.6](#):

```

1  namespace Chapterr9.Models
2  {
3      0 references
4      public interface IMobilePlan
5      {
6          0 references
7          IConnectionSpeed ConnectionSpeed { get; set; }
8          0 references
9          ITextMessage TextMessage { get; set; }
10     }
11 }

```

Figure 9.6: IMobilePlan interface

As was said regarding the connection speed and text message classes, all the classes that implement the **IMobilePlan** interface are not responsible for specifying any values to the corresponding

properties because this should be by the future factory class. Considering the requirements include two different mobile plans, it is necessary to create the concrete class for each type, implementing the **IMobilePlan** interface in the **PostPaidMobilePlan** class, as shown in [Figure 9.7](#):

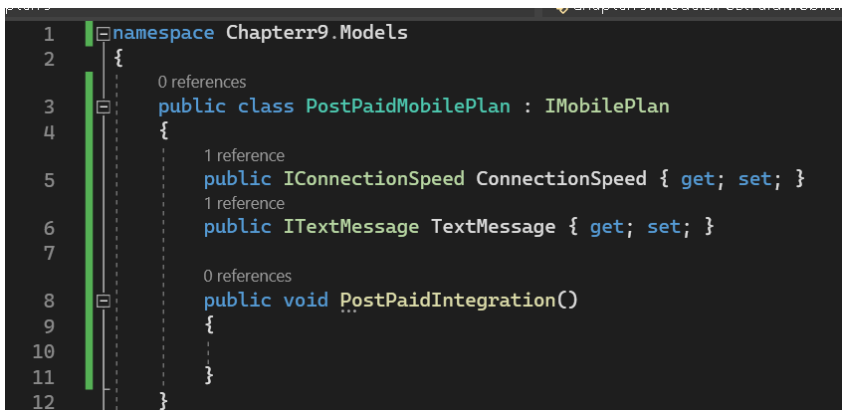


Figure 9.7: *PostPaidMobilePlan* class

Once each mobile plan has its own underlying class, it is possible from this point to specify additional custom methods that can be used if necessary for each class. With this approach, even though there is a factory to create each type, each class is still not dependent on the other. This follows good object-oriented programming practices in general.

The next step is to create the **PrePaidMobilePlan** class for the pre-paid mobile plan, which is achieved by the code demonstrated in [Figure 9.8](#):


```

1  namespace Chapter9.Models
2  {
3      0 references
4      public class PrePaidMobilePlan : IMobilePlan
5      {
6          1 reference
7          public IConnectionSpeed ConnectionSpeed { get; set; }
8          1 reference
9          public ITextMessage TextMessage { get; set; }
10
11      0 references
12      public void PrePaidIntegration()
13      {
14      }
15  }

```

Figure 9.8: PrePaidMobilePlan class

As you can realize in [Figure 9.8](#), the pre-paid mobile plan class contains a method called **PrePaidIntegration**, which is exclusive to that class. This is an advantage of using the Factory Method pattern rather than the Abstract Factory method pattern in specific scenarios, as it is possible to specialize each class more efficiently. However, the current pattern still requires the creation of factories, one for each mobile plan type, in order to achieve the purpose of encapsulating the creation of the objects from the consumer application. In this case, you need to create the base factory class that will be shared for all the factories, as shown in [Figure 9.9](#):

```

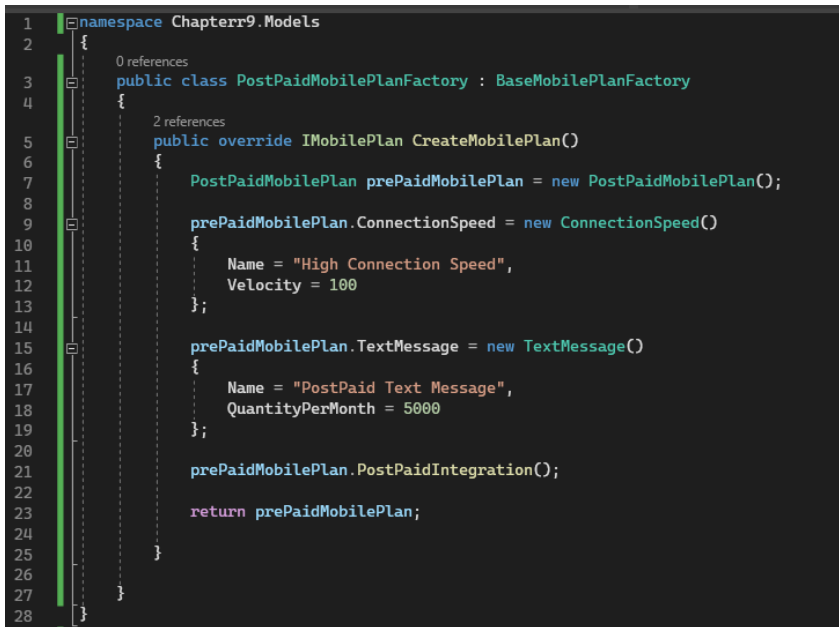
1  namespace Chapter9.Models
2  {
3      0 references
4      public abstract class BaseMobilePlanFactory
5      {
6          0 references
7          public IMobilePlan GetMobilePlan()
8          {
9              IMobilePlan mobilePlan = this.CreateMobilePlan();
10             return mobilePlan;
11         }
12
13         1 reference
14         public abstract IMobilePlan CreateMobilePlan();
15     }

```

Figure 9.9: BaseMobilePlanFactory class

Following the best practices stated by the Factory Method pattern, the base factory class must use the abstract modifier, and the **GetMobilePlan** method must refer to a method responsible for creating the mobile plan instance. You may see the same pattern used in legacy projects or external libraries. This characteristic indicates an intention to delegate the creation of the instances to the factory's subclasses, indicating the main difference between this pattern and the Abstract Factory one.

Once the base factory is created, it is time to create the individual factories. The individual factories will inherit from the base class, making all the necessary custom operations required for the individual types, as demonstrated in the code illustrated in [Figure 9.10](#):

A screenshot of a code editor showing the implementation of the `PostPaidMobilePlanFactory` class. The code is in C# and is part of the `Chapterr9.Models` namespace. The class inherits from `BaseMobilePlanFactory`. It overrides the `CreateMobilePlan()` method. Inside this method, a `PostPaidMobilePlan` object is created, its `ConnectionSpeed` property is set to a new `ConnectionSpeed` object with the name "High Connection Speed" and velocity 100, and its `TextMessage` property is set to a new `TextMessage` object with the name "PostPaid Text Message" and quantity per month 5000. Finally, the `PostPaidIntegration()` method is called on the `prePaidMobilePlan` object, and the object is returned.

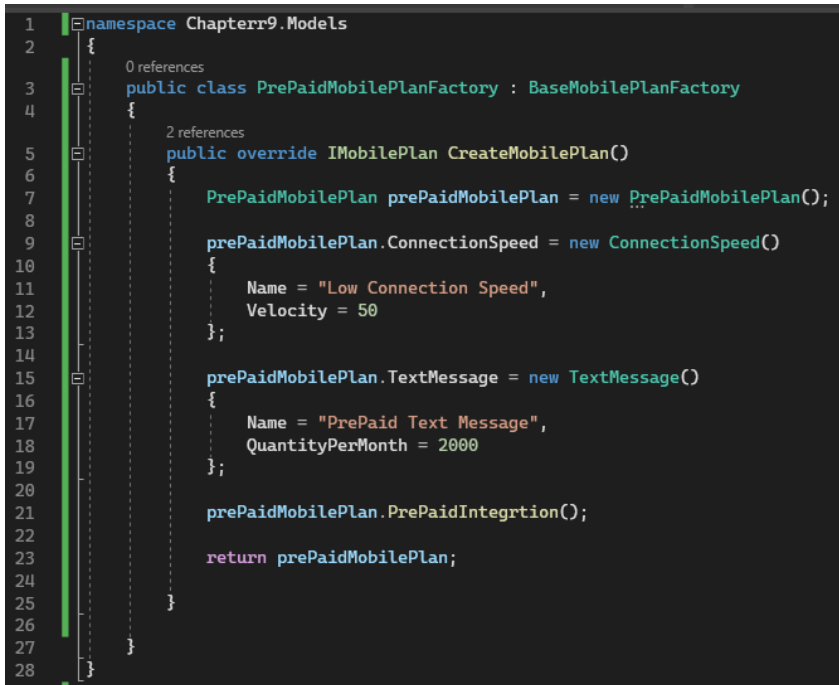
```
1 namespace Chapterr9.Models
2 {
3     public class PostPaidMobilePlanFactory : BaseMobilePlanFactory
4     {
5         public override IMobilePlan CreateMobilePlan()
6         {
7             PostPaidMobilePlan prePaidMobilePlan = new PostPaidMobilePlan();
8
9             prePaidMobilePlan.ConnectionSpeed = new ConnectionSpeed()
10            {
11                Name = "High Connection Speed",
12                Velocity = 100
13            };
14
15            prePaidMobilePlan.TextMessage = new TextMessage()
16            {
17                Name = "PostPaid Text Message",
18                QuantityPerMonth = 5000
19            };
20
21            prePaidMobilePlan.PostPaidIntegration();
22
23            return prePaidMobilePlan;
24        }
25    }
26 }
27
28 }
```

Figure 9.10: PrepaidMobilePlanFactory class

The **PrePaidMobilePlanFactory** class inherits from the base factory abstract class and overwrites the method responsible for creating the mobile plan, specifying the custom values for the connection speed and text message properties according to the requirements. Additionally, the method calls the

PrePaidIntegration method for running custom operations regarding the creation of the instance. The integration method does not have any implementation for demonstration purposes, but real scenarios may require extra operations associated with each factory.

Finally, as stated by the requirements, the next step is to create the underlying factory class for the post-paid mobile plan (**PostPaidMobilePlanFactory**), following the same pattern as the previous code, as shown in [Figure 9.11](#):



```
1 namespace Chapter9.Models
2 {
3     0 references
4     public class PrePaidMobilePlanFactory : BaseMobilePlanFactory
5     {
6         2 references
7         public override IMobilePlan CreateMobilePlan()
8         {
9             PrePaidMobilePlan prePaidMobilePlan = new PrePaidMobilePlan();
10
11             prePaidMobilePlan.ConnectionSpeed = new ConnectionSpeed()
12             {
13                 Name = "Low Connection Speed",
14                 Velocity = 50
15             };
16
17             prePaidMobilePlan.TextMessage = new TextMessage()
18             {
19                 Name = "PrePaid Text Message",
20                 QuantityPerMonth = 2000
21             };
22
23             prePaidMobilePlan.PrePaidIntegration();
24
25             return prePaidMobilePlan;
26         }
27     }
28 }
```

Figure 9.11: *PostpaidMobilePlanFactory* class

With this class, the Factory Method pattern is fully implemented, following all the business requirements specified at the beginning of this section. The last thing that needs to be done is to test the factory classes in a client application and check the results. In a Console application, you can create an instance of each factory class and verify the output results, as seen in [Figure 9.12](#):

```

0 references
static void Main(string[] args)
{
    BaseMobilePlanFactory prePaidFactory = new PrePaidMobilePlanFactory();
    IMobilePlan prePaidMobilePlan = prePaidFactory.GetMobilePlan();

    BaseMobilePlanFactory postPaidFactory = new PostPaidMobilePlanFactory();
    IMobilePlan postPaidMobilePlan = prePaidFactory.GetMobilePlan();

    Console.WriteLine("*****POST-PAID MOBILE PLAN*****");
    Console.WriteLine("Connection speed:" + postPaidMobilePlan.ConnectionSpeed.Velocity.ToString());
    Console.WriteLine("Text Message:" + postPaidMobilePlan.TextMessage);

    Console.WriteLine("*****POST-PAID MOBILE PLAN*****");
    Console.WriteLine("Connection speed:" + prePaidMobilePlan.ConnectionSpeed.Velocity.ToString());
    Console.WriteLine("Text Message:" + prePaidMobilePlan.TextMessage);
}

```

Figure 9.12: Factory class instances

An instance of the mobile is retrieved by using the **GetMobilePlan** method provided by each individual factory class, hiding all the aspects of the implementation from the client application, reaching the main purpose of the Factory Method pattern. As a final step for the client application, print the properties of each plan in the console, as shown in [Figure 9.13](#):

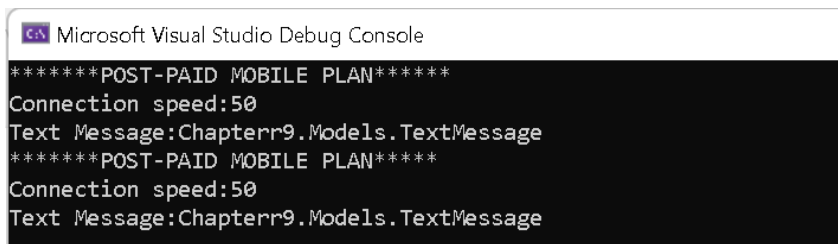
```

1  using Chapter9.Models;
2
3  namespace Chapter9
4  {
5      0 references
6      internal class Program
7      {
8          0 references
9          static void Main(string[] args)
10         {
11             BaseMobilePlanFactory prePaidFactory = new PrePaidMobilePlanFactory();
12             IMobilePlan prePaidMobilePlan = prePaidFactory.GetMobilePlan();
13
14             BaseMobilePlanFactory postPaidFactory = new PostPaidMobilePlanFactory();
15             IMobilePlan postPaidMobilePlan = prePaidFactory.GetMobilePlan();
16         }
17     }

```

Figure 9.13: Factory method output

Suppose you run the client application with the given implementation. In that case, the console displays the difference between the pre-paid and post-paid mobile plans, demonstrating that the implementation works as expected and keeps all the complexity of the implementation in the **Factory** class instead of the consumer client application, as shown in [Figure 9.14](#):



```
Microsoft Visual Studio Debug Console
*****POST-PAID MOBILE PLAN*****
Connection speed:50
Text Message:Chapterr9.Models.TextMessage
*****POST-PAID MOBILE PLAN*****
Connection speed:50
Text Message:Chapterr9.Models.TextMessage
```

Figure 9.14: Factory method result

As illustrated in this chapter, the implementation of the Factory Method pattern requires less effort than the Abstract Factory pattern, being possible to combine simplicity with a fantastic architecture that allows us to encapsulate the full implementation of significant operations in abstract classes, avoiding access violation and unexpected behavior in real scenarios.

New features in C# 11

The C# language always evolves following the feedback from technical communities, IT professionals, and companies, mainly after the .NET platform became an open-source project since the first .NET Core version. Each new release contains relevant enhancements that help developers to create more robust applications in terms of performance and coding style.

While this book has the primary purpose of teaching Design Patterns using the C# language, each chapter contains an overview of new features introduced in .NET 7 and C# 11 to combine the knowledge of Design Patterns with the most modern capabilities in the programming language that is used along with the code samples in this book.

Given that, the following sections will show give you a high-level overview of the main features traduced to the C# language since version 11. It is important to note that, considering the C# language and the .NET platform are constantly changing for the better, it is recommended to check the official Microsoft documentation to get more updates if they are available after the time you read this book.

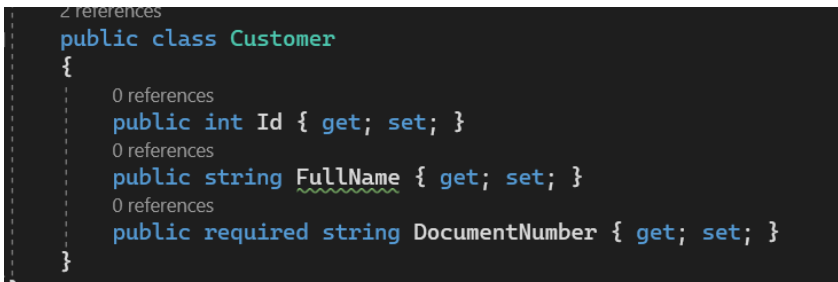
Required fields

When a new instance of any object is created, it is usually necessary

to include validation for required fields in the class's constructor. In previous versions of C# language, this process is quite tricky to make as it needs to be done by abstract classes and any other child class by inheritance.

The C# 11 version introduced the required modifier to be used in properties, which can be used to force specific fields to be populated when a new instance of an object is created. This new feature applies to classes, structs, and record types.

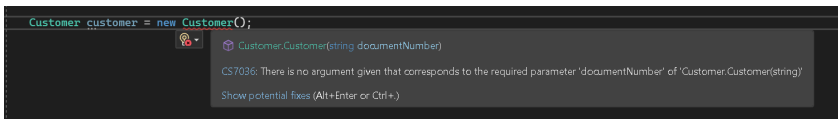
As you can see in [Figure 9.15](#), the **DocumentNumber** property is marked as required in the **Customer** model:



```
2 references
public class Customer
{
    0 references
    public int Id { get; set; }
    0 references
    public string FullName { get; set; }
    0 references
    public required string DocumentNumber { get; set; }
}
```

Figure 9.15: Required modifier

When the field is marked as required, any attempt to create an instance of the underlying object results in compiler errors, as seen in [Figure 9.16](#):



```
Customer customer = new Customer();
```

Customer.Customer(string documentNumber)

CS7036: There is no argument given that corresponds to the required parameter 'documentNumber' of 'Customer.Customer(string)'

Show potential fixes (Alt+Enter or Ctrl+.)

Figure 9.16: Compiler error

In order to resolve this issue, it is necessary to modify the constructor of the **Customer** class to receive a document number as input to be used to initialize the corresponding property using the **SetsRequiredMembers** attribute, as seen in [Figure 9.17](#):

```

3 references
public class Customer
{
    [SetsRequiredMembers]
    1 reference
    public Customer (string documentNumber)
    {
    }
    0 references
    public int Id { get; set; }
    0 references
    public string FullName { get; set; }
    0 references
    public required string DocumentNumber { get; set; }
}

```

Figure 9.17: Customer constructor

After modifying the `Customer` constructor method, you can pass the underlying document value in the constructor, which is going to resolve the compiler issue, as seen in [Figure 9.18](#):

```

Customer customer = new Customer("123456");

```

Figure 9.18: Customer initialization

The required modifier can be used for multiple fields in the same class, struct, or record, and it is quite helpful to avoid the overuse of conditional statements in the code or the use of reflection in C# language to achieve similar goals.

UTF-8 string literals

C# 11 introduced the possibility of using the `u8` suffix on a string to determine that the string should use UTF-8 encoding. This is helpful when several routines in the application require this type of encoding as part of a specific protocol. You can specify the suffix as shown in [Figure 9.19](#):

```

var myString = "my string"u8;

```

Figure 9.19: UTF-8 suffix

It is essential to highlight that Strings in .NET use the UTF-16 encoding. The use of UTF-8 string literals can not be used as a default value for the optional parameter as they are runtime constants.

Newlines in string interpolations

Additionally to the possibility of converting strings to UTF-8, C# 11 introduced a better way to handle newlines for string interpolations. With the new syntax, it is possible to specify switch cases and other commands inside the string interpolation, as seen in [Figure 9.20](#):

```
string dayOfWeekMessage = $"Todah is {  
    DateTime.Now.DayOfWeek switch  
    {  
        DayOfWeek.Monday => "Monday. Lets get back to work",  
        DayOfWeek.Friday => "Friday. Lets celebrate"  
    }  
}";
```

Figure 9.20: Newlines in string interpolation

This new feature improves the readability of expressions that involves pattern matching and reduces the number of lines of code needed for string interpolation.

List patterns

From C# 11, it is possible to match and compare lists with a sequence of patterns, as seen in [Figure 9.21](#):

```
string[] arrayStrings = { "Jack", "Paul", "Mandy" };  
  
Console.WriteLine(arrayStrings is ["Jack", "Paul", "Mandy"]); // True  
Console.WriteLine(arrayStrings is ["Jack", "Paul", "Silver"]); // False  
Console.WriteLine(arrayStrings is ["Jack", "Paul", "Mandy", "Silver"]); // False
```

Figure 9.21: List patterns

As shown in [Figure 9.21](#), a specific pattern is matched only when all elements are matched between the pattern and the list, following their corresponding order.

Raw string literals

Raw string literals in C# language are used to store arbitrary text, such as new lines, quotes, and many other special characters. Version 11 of C# language allows specifying double quotes within the content without having to escape the characters, as demonstrated in *Figure 9.22*:

```
string rawString = """ This is a string with "quotes" in the content""";  
Console.WriteLine(rawString);
```

Figure 9.22: Raw string literals

This new feature increases the code's readability, similar to what is done in other programming languages, such as Python.

Conclusion

As you learned in this chapter, the factory method pattern is beneficial for managing the creation of complex objects with simplicity and efficiency, using the best practices regarding encapsulation and polymorphism. Considering that the pattern extensively uses interfaces and abstract classes, it is possible to have a consistent architecture across the project taking all the benefits of mocking objects for testing and other purposes. This pattern reduces the dependence between classes and gives us the potential to extend functionality without causing any impact on legacy code.

In this chapter, you have learned how to implement the factory method pattern using the C# language in a real scenario, and the ability to understand the general central point of design patterns is demonstrated. This supports the ability to use general solutions for common problems in software development in terms of implementation and software architecture.

In the next chapter, you will have the chance to continue your journey into the design patterns using the C# language and .NET platform, learn the Adapter pattern with Entity Framework Core, and apply all the general knowledge you have received in the first chapters of this book.

Points to remember

- The factory method pattern is much more straightforward to use and implement than the abstract factory one, as it primarily aims to create a single object as a factory.
- The C# language fully supports the factory method pattern because it is based on interfaces and abstract classes, standard features in the object-oriented programming paradigm.

Multiple choice questions

- 1. What is the main difference between abstract factory and factory method patterns?**
 - A. The abstract factory creates factories for the subclasses, and the factory method is only for the main class.
 - B. There is no difference between them
 - C. The abstract factory creates more simple objects than the factory method pattern.
 - D. None of them
- 2. Which class creates the subclasses' instances in the factory method pattern?**
 - A. The main factory.
 - B. The underlying subclasses.
 - C. The client application.

Answer

	A	
	B	

Questions

1. Explain the main purpose of the factory method pattern.
2. Why is the use of interfaces necessary in the factory method pattern?
3. According to what you have learned in this chapter, create a

hypothetical scenario where the factory method could be applied, apart from the example given in this chapter.

Key terms

- **Factory:** Logical implementation to help in the construction of complex objects.
- **Inheritance:** It allows us to create a correlation between classes, creating a parent class with a generic implementation to be used by other child classes.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Adapter Pattern with Entity Framework Core

Introduction

Integration across various systems represents a quintessential scenario in the landscape of contemporary applications. Frequently, a single organization is required to utilize an array of systems to cater to the exhaustive needs of their business processes. These processes are often supported by diverse applications, and it's quite likely that the software employed does not adhere to a uniform set of interface specifications and protocols.

This is where the robustness of the adapter pattern comes into play. It aids in architecting a cohesive interface, facilitating seamless communication between systems that do not natively support compatible inputs and outputs. Importantly, this obstacle is not confined to systems developed by different entities but can also be observed between legacy and newly initiated projects within the same organization.

Mastering the adapter pattern equips you with the ability to devise an elegant and efficient architectural blueprint. This blueprint can be used to construct interfaces between incompatible systems, while simultaneously applying the gold-standard practices of SOLID principles. An example of such a principle is the open-close principle, which encourages the extension of integration among components without disrupting the original implementation of the components participating in the process. Understanding the adapter design pattern not only provides you with a beneficial tool in your programming toolkit, but also allows you to illustrate adept object-oriented programming practices when developing, modifying software, or creating application architecture.

Through this chapter, you will delve into the intricacies of the adapter pattern using the C# language. The principles will be elucidated in a practical, step-by-step sample, paving the way for an enriched understanding and application of this pivotal pattern.

Structure

In this chapter, we will discuss the following topics:

- Adapter design pattern definition
- Adapter design pattern implementation
- Code samples and practical exercises

Objectives

In this chapter, our objectives are multifaceted. Firstly, we aim to deepen your understanding of system integration, a crucial requirement in today's diverse application landscape.

Secondly, we seek to familiarize you with the adapter pattern, a powerful tool in reconciling incompatible interface specifications and protocols across various systems. In doing so, we'll delve into the creation of friendly interfaces that facilitate seamless communication between disparate systems, be they from different companies or within the same organization.

Thirdly, we intend to provide a thorough exploration of how the adapter pattern can uphold SOLID principles, particularly the open-close principle, to allow for effective, non-disruptive integration.

Lastly, we'll employ the C# language to unpack these concepts through a practical, step-by-step example, thus demonstrating the application of the adapter pattern in real-world scenarios.

By the end of this chapter, you should be able to competently apply the adapter pattern and object-oriented programming practices to develop efficient system interfaces, whether you're working on new software or modifying an existing application architecture.

Adapter pattern definition

The adapter pattern holds a prominent position in the realm of software development, acting as a key facilitator for integrating diverse systems. Frequently, developers grapple with compatibility issues among various components. Moreover, a component often designed for a specific purpose may present additional requirements when integrated with others. The aim is to ensure a seamless integration while preserving the integrity of the original systems, circumventing disruptive changes to both the source and target components.

In essence, any system that integrates with external resources implicitly utilizes the adapter pattern. The data format or method signatures between the source and target can diverge significantly, especially when connections to databases are involved. Every database has its unique data storage method, but libraries in C# or other languages typically ease communication between the database and the application. Given that databases interpret data using specific protocols and formats while C# expects output in the form of classes, objects, or primitive types, an adapter becomes indispensable.

An adapter serves to bridge the gap between natively incompatible interfaces, enabling the utilization of their functionalities while keeping the original components intact. For instance, if a system needs to process data in the JSON format, but a legacy project provides data in XML format, it is highly advisable to retain the XML format in the original system. Instead of disrupting the existing interface, an adapter can be created to convert the XML data from the legacy system into the required JSON format. Integrations often involve third-party software, whose systems may not be open to modifications or changes in the original source code. In such cases,

any alterations to legacy code can escalate software development costs and pose substantial challenges in terms of code interpretation, adaptability, and potential unfamiliarity with the legacy technology amongst the development team.

Given that the adapter must be constructed without impacting the involved components, the pattern necessitates the creation of an adapter class. This class is responsible for converting the source format into the target class format and vice versa. This practice is an integral aspect of integration and aligns with the open-close principle, a crucial tenet of the SOLID principles. Generally, the adapter design pattern have the structure represented in the following diagram:

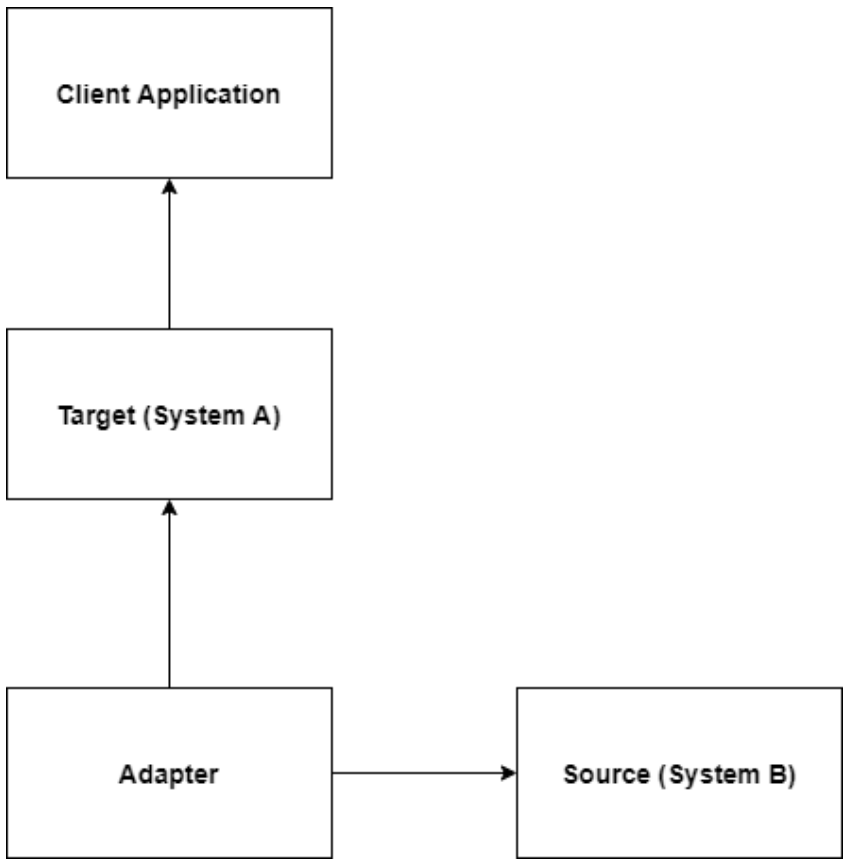


Figure 10.1: Adapter pattern diagram

While the complexity of system integration can vary significantly, the fundamental structure of the adapter pattern remains consistent. It extends the interface, typically to reconcile communication between the source and target, which often suffer from incompatibility in their input and output formats. This design pattern offers the flexibility to have multiple connectors and adapters, all operating independently of each other. This independent operation drastically minimizes the risk of incompatibility errors surfacing in production environments, ensuring smoother and more reliable system performance.

Adapter pattern implementation

As with the exploration of other design patterns in previous chapters, gaining a comprehensive understanding of the adapter pattern demands a hands-on approach. We will implement a project that mirrors a real-world scenario. Consider an online store with a legacy system, developed two decades ago, that furnishes customer information in XML format. A popular format twenty years ago, the XML-based system has been maintained due to its stability and the store's enduring contract with a software provider. As a result, it is crucial for the current interfaces to remain intact to ensure continued cooperation with existing providers.

Contrastingly, the company's other systems employ modern technologies like .NET Core and the C# language. All the APIs exposed by these newer systems employ the JSON format and REST technology. Thus, the output of the legacy system is not directly compatible with the new systems which primarily utilize JSON-formatted messages.

To resolve this predicament, several options emerge. One could be replacing the old system with a newer one to unify the technology across the company's systems. However, this path would demand substantial time and resources from the development team - assets that are often allocated to other projects. Given the constraints of deadlines, costs, and general project considerations, this option is unlikely to be greenlighted by the decision-making management.

An alternative could involve modifying the legacy system to provide responses in the JSON format, thereby ensuring compatibility with

the other systems. However, the scenario clearly stipulates that the system is not open to modification. The lack of access to the source code and the provider's refusal to permit alterations to the product core rule out this option.

In such a context, our most viable course of action is to create an adapter. This tool can adeptly manage the discrepancy between the formats of the new and legacy systems. Implementing this approach allows us to construct a wrapper capable of processing the XML format and transforming the response into JSON, without necessitating any alterations to the systems involved.

In line with the structure proposed by the adapter pattern and the requirements of the given scenario, the solution's implementation should specify an adapter. This adapter would be tasked with converting the legacy system's response into a format that the new system can interpret. This strategy opens up the possibility of connecting the application with various other systems, should their data need translating into a format the target application can understand.

As evident in this solution type, it's common to find similar approaches in libraries facilitating database integration, such as Entity Framework and Hibernate. Essentially, these libraries establish a wrapper to connect two distinct systems (application and database). Standard ADO.NET technology offers common interfaces that enable any database providers to develop their own wrappers for the C# language. As you may have noted, some classes in ADO.NET feature the suffix **adapter**, signifying their adherence to the same concept as the adapter design pattern.

In the context of our sample scenario, the architecture of the adapter pattern would adopt the structure delineated in the forthcoming [Figure 10.2](#):

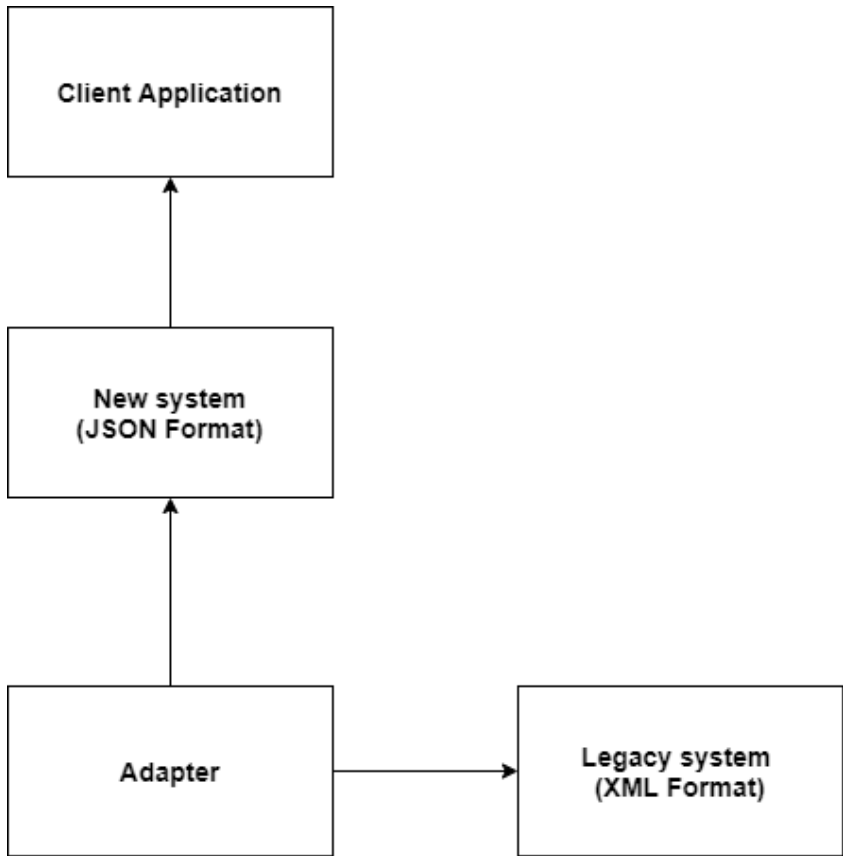


Figure 10.2: Adapter pattern implementation diagram

This diagram, akin to the one presented in [Figure 10.1](#), distinctively labels components in accordance with the business requirements. When defining software architecture, it's advisable to first outline the details of the intended pattern before finalizing the architecture. A design pattern implemented in any programming language should be readily identifiable to both current and future developers responsible for software maintenance. Therefore, it's standard practice to reference the intended design patterns in the names of related classes and interfaces.

In real-world situations, maintaining and identifying design patterns can become challenging, particularly when they're interwoven with other patterns and lack clear, descriptive naming. Complications also arise when an inappropriate pattern is used to address a

specific problem. It's not uncommon to find projects where a design pattern is implemented, not due to project necessity, but because a developer wished to learn about the pattern. However, the decision to deploy a specific design pattern should always be guided by the business requirements and the problem at hand.

With respect to our hypothetical scenario involving the adapter pattern, our next course of action is to generate the code necessary for implementing this pattern. Given that the legacy system is responsible for providing customer information, we need to develop classes to mimic its behavior. The first class to be created is the **Customer** class, utilized by the legacy project, as illustrated in the upcoming [Figure 10.3](#):

```
1  using System.Xml.Serialization;
2
3  namespace DesignPatternsBookChapter10
4  {
5      [Serializable]
6      public class Customer
7      {
8          [XmlAttribute]
9          public int Id { get; set; }
10         [XmlAttribute]
11         public string Name { get; set; }
12         [XmlAttribute]
13         public string Address { get; set; }
14         [XmlAttribute]
15         public string City { get; set; }
16         [XmlAttribute]
17         public string Country { get; set; }
18         [XmlAttribute]
19         public string DocumentNumber { get; set; }
20         [XmlAttribute]
21         public DateTime BirthDay { get; set; }
22     }
23 }
```

Figure 10.3: Customer class

As depicted in the preceding [Figure 10.3](#), all the class properties bear XML attributes. This allows for serialization to the XML format and effectively emulates the behavior of the legacy system. Subsequently, we need to construct a logic class responsible for returning a customer list in the XML format, serving demonstration purposes. In an actual scenario, this implementation would be superfluous, as the legacy project would be delivering the data. However, if you aim to fully replicate the adapter pattern example in this chapter, creating these classes is a requisite. This can be achieved within a standard Console application, which is the approach we've adopted in this section. The implementation of the customer logic class will be demonstrated in the upcoming [Figure 10.4](#):

```
1 using System.Xml;
2 using System.Xml.Serialization;
3
4 namespace DesignPatternsBookChapter10
5 {
6     1 reference
7     public class CustomerLogic
8     {
9         private List<Customer> customers;
10        0 references
11        public CustomerLogic()
12        {
13            customers = new List<Customer>();
14            customers.Add(new Customer() { Id = 1, Name = "Customer 1", City = "New York" });
15            customers.Add(new Customer() { Id = 2, Name = "Customer 2", City = "Los Angeles" });
16            customers.Add(new Customer() { Id = 3, Name = "Customer 3", City = "Las Vegas" });
17            customers.Add(new Customer() { Id = 4, Name = "Customer 4", City = "Sao Paulo" });
18            customers.Add(new Customer() { Id = 5, Name = "Customer 5", City = "Dublin" });
19        }
20
21        0 references
22        public virtual string GetCustomers()
23        {
24            var xmlSerializer = new XmlSerializer(customers.GetType());
25            using (var stream = new StringWriter())
26            {
27                using (var xmlWriter = XmlWriter.Create(stream))
28                {
29                    xmlSerializer.Serialize(xmlWriter, customers);
30                    return stream.ToString();
31                }
32            }
33        }
34    }
35 }
```

Figure 10.4: Customer logic class

The constructor of the class generates a list of five customers, and the **GetCustomers** method employs the virtual modifier, allowing for its implementation to be overridden if necessary. Initially, this method serializes the customer list to the XML format, ideally mimicking the legacy project's behavior in our scenario. Having created all classes pertaining to the legacy software, we now turn

our attention to defining the necessary interfaces and classes to implement the adapter pattern. The first contract to be constructed is the **ICustomer** interface, which abstracts the operations performed by the **CustomerLogic** class, as depicted in the upcoming *Figure 10.5*:

```
1 namespace DesignPatternsBookChapter10
2 {
3     0 references
4     public interface ICustomer
5     {
6         0 references
7         string GetCustomers();
8     }
9 }
```

Figure 10.5: Customer interface

Subsequently, we must establish the pattern's primary class, which references both the legacy **CustomerLogic** class and the newly formed **ICustomer** interface. This can be seen in the ensuing *Figure 10.6*:

```
1 using Newtonsoft.Json;
2 using System.Xml;
3
4 namespace DesignPatternsBookChapter10
5 {
6     0 references
7     public class CustomerAdapter : CustomerLogic, ICustomer
8     {
9         3 references
10        public override string GetCustomers()
11        {
12            string originalXml = base.GetCustomers();
13            XmlDocument xmlDoc = new XmlDocument();
14            xmlDoc.LoadXml(originalXml);
15
16            JsonSerializerSettings jsonSettings = new JsonSerializerSettings();
17            jsonSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
18
19            var customers = JsonConvert.SerializeObject(xmlDoc, jsonSettings);
20
21            return customers;
22        }
23    }
24 }
```

Figure 10.6: Customer adapter class

As illustrated in the previous [Figure 10.6](#), the **CustomerAdapter** class inherits from the **CustomerLogic** class and overrides the **GetCustomers** method. This method now includes an implementation capable of converting the XML content into the JSON format with respect to the customer list information. To implement the above code, it is essential to install the **Newtonsoft.Json** package to access enhanced JSON serialization options. In the **NuGet** package window, search for this package and proceed with the installation as demonstrated in the forthcoming [Figure 10.7](#):

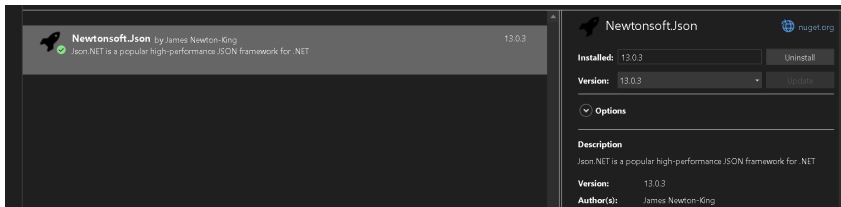


Figure 10.7: Newtonsoft NuGet package

Given that the comprehensive example of the adapter pattern has been implemented, it's now feasible to test and ascertain if the entire structure functions as anticipated. To validate the **CustomerAdapter** class, instantiate a new object within the Console application and invoke the **GetCustomers** method. This process is displayed in the upcoming [Figure 10.8](#):

```
1 namespace DesignPatternsBookChapter10
2 {
3     0 references
4     internal class Program
5     {
6         0 references
7         static void Main(string[] args)
8         {
9             CustomerAdapter customerAdapter = new CustomerAdapter();
10            var customers = customerAdapter.GetCustomers();
11
12            Console.WriteLine(customers);
13
14            Console.ReadLine();
15        }
16    }
```

Figure 10.8: Adapter pattern client

Despite its apparent simplicity in comparison to other patterns, requiring the creation of only one class to mediate communication between distinct systems, the adapter pattern is both powerful and secure. It facilitates system and component integration that don't share similar interfaces, enabling functionality extension without altering source code. Each design pattern is conceived with the aim to address a common problem in software development. Consequently, numerous companies and projects encounter similar issues, making these patterns widely applicable.

With the understanding gained from this chapter, you are encouraged to employ the adapter pattern in real-life projects. However, always remember to use the adapter pattern judiciously, basing decisions on the specific needs of each project. This nuanced approach will maximize the benefit of applying this pattern, promoting efficiency and adaptability in your software development processes.

Conclusion

This chapter has elucidated the power of the adapter pattern as a solution for bridging systems that don't inherently share compatible input and output formats yet require interaction and communication. Renowned as one of the most frequently implemented design patterns, the adapter pattern significantly enhances communication between software applications grounded in object-oriented programming.

Underpinning this pattern is the open-closed principle from the SOLID principles set, promoting the practice of extending functionality without modifying the behaviour of the involved components. Thus, source and target classes or components can maintain their current interfaces while providing interface extensibility to accommodate data with varying content and format. Here, the adapter pattern upholds the responsibility of data conversion between two components through a solitary adapter class.

Through this chapter, you've not only learned how to apply the adapter pattern in a practical scenario using C#, but also gained an appreciation for the core purpose of design patterns more broadly.

That is, they offer solutions to software development challenges that align with object-oriented programming's best practices and are widely embraced across industry.

As you progress to the next chapter, you'll continue your deep dive into design patterns with C# and the .NET platform. This exploration will include a study of the composite pattern, further augmenting the comprehensive knowledge you've accrued in the book's preceding chapters.

Points to remember

- Although the adapter pattern ranks among the simpler design patterns, its application wields tremendous power in integrating disparate systems.
- The implementation of the adapter pattern does not necessitate the creation of numerous classes or intricate levels—only the crafting of the adapter class is essential.
- This pattern finds extensive utilization in libraries tasked with database interactions and manipulation of external resources, underlining its practicality and effectiveness.

Multiple choice questions

1. Which of the following does not correctly characterize the adapter pattern?
 - A. The adapter pattern assists in integrating disparate systems
 - B. The adapter pattern is complex and difficult to implement
 - C. Database integrations frequently employ the adapter pattern
 - D. None of the above
2. In which class should the logic governing system integration reside?
 - A. Adapter class
 - B. Client application
 - C. Abstract class within the system
 - D. Source class

Answers

	3	
	2	

Questions

1. Elucidate the principal objective of the adapter pattern.
2. Drawing upon the knowledge gleaned from this chapter, devise a hypothetical context, apart from the presented example, where the adapter pattern might be effectively deployed.
3. Expound on the advantages of employing the adapter pattern.
4. Are there any ongoing projects in which you are engaged that incorporate the adapter pattern?

Key terms

- **Client Application:** This is the most abstract layer of a system, typically encompassing the user interface.
- **System Integration:** This is the process of enabling communication among various components, employing specified protocols.
- **API:** An Application Programming Interface is generally designed to facilitate integration among disparate systems.
- **Cache:** In the realm of software development, caching involves storing data in a location that offers superior performance compared to the original source.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Composite Pattern with ASP.NET MVC

Introduction

Managing hierarchical and recursive structures within any programming language can be a daunting task due to issues of performance, design complexity, and intricate logic. The composite design pattern, however, simplifies this challenge, providing an elegant solution to handle scenarios where multiple nested levels of an object are needed. This ensures that the underlying classes remain unentangled, adhering to the principle of independence.

This pattern is closely tied to the **Liskov Substitution Principle**, a concept we previously discussed in this book. When delving into design patterns, one can appreciate the interconnectedness of different aspects of software development, ranging from SOLID principles and object-oriented programming, to software architecture and quality assurance.

Understanding the composite pattern is more than a learning milestone; it's an opportunity to gain a deeper insight into the key principles of inheritance. Moreover, it unravels the core purpose of

properly utilizing interfaces. It facilitates the exploration of advanced concepts in software architecture such as extensibility, maintainability, and testability.

In this chapter, we go beyond theoretical knowledge, as you will get hands-on experience by applying the composite pattern to a practical example. We will walk you through the process of creating a real-world application using C#, step by step. You will not only understand the mechanics of the composite design pattern, but also master its implementation.

Structure

In this chapter, we will discuss the following topics:

- Composite design pattern definition
- Composite design pattern implementation
- Code samples and practical exercises

Objectives

This chapter will deepen your understanding of the composite pattern, a design pattern essential for handling hierarchical and recursive structures in programming. It will enhance your problem-solving and software design proficiency by teaching you the theoretical aspects, real-world applicability, and situations for effective employment of the composite pattern.

You will also learn about the SOLID principles, five key principles of object-oriented programming and design, and how the composite pattern embodies them to create robust, scalable, and maintainable code. Lastly, you will have the opportunity to apply the composite pattern in a real-world programming scenario, following a step-by-step guide to solve a complex problem, which will reinforce your understanding and confidence in using the composite pattern in future coding projects.

Composite pattern definition

The composite pattern articulates that a collection of objects should behave in the same way as any single object within the group. Essentially, it blurs the lines between individual and composite

objects, ensuring that operations can be called on a component whether it is a single object or a group of objects. This unifying interface allows a client to treat individual objects and compositions uniformly.

In the real-world, the composite pattern mirrors a tree-like structure, where an element can have numerous child elements of the same type, thereby establishing a parent-child relationship. This facilitates invoking methods from a child element as seamlessly as you would from its parent elements. A visual representation of this structural hierarchy is provided in [Figure 11.1](#):

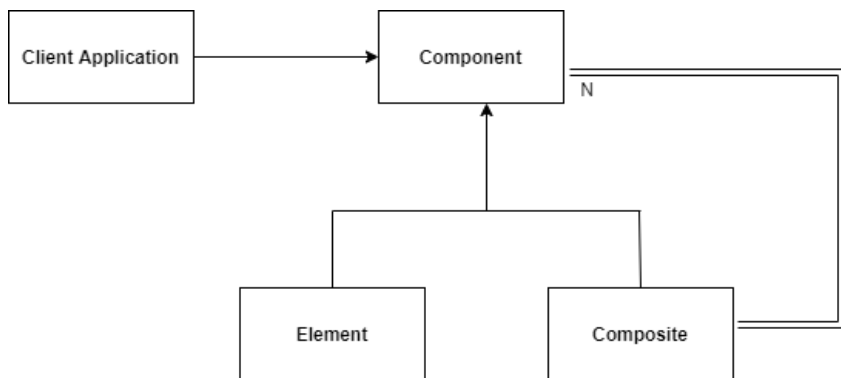


Figure 11.1: Composite pattern diagram

In this particular context, a component may either be a single element or a parent element that encompasses multiple sub-elements. The client application is kept oblivious to an element's composition, thus treating all elements uniformly. This abstraction conceals the distinction between parent and child elements. As previously mentioned, all the components and classes involved in this process are of the same type or share a common interface. This approach utilizes the concept of encapsulation to shield the intricacies of the implementation from the client application. It's a practical demonstration of the Liskov Substitution Principle explained in [Chapter 3, Basic Concepts of Object-Oriented Programming](#).

The composite pattern can also be represented as a hierarchical diagram for better visual comprehension. This representation is more reflective of the scenarios where the pattern is frequently

applied. As depicted in [Figure 11.2](#), a specific element can have several sub-elements, even though they are of the same kind. This illustrative approach not only simplifies understanding but also makes the pattern's application more intuitive:

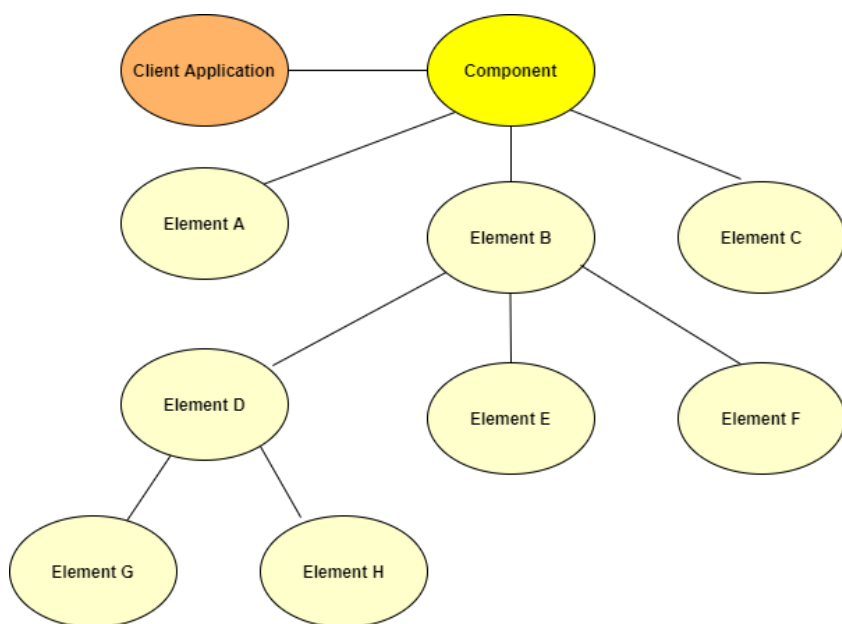


Figure 11.2: Hierarchical representation

This tree-like structure, a characteristic feature of the composite pattern, pervades various applications, including menu interface components across web, mobile, and desktop platforms. Due to the pattern's analogous architecture to a tree, a single element in the diagram is often referred to as a **leaf**, while elements containing sub-elements are known as **composites**. You will find it easier to grasp the composite pattern's concept and usage through a real-world example. The upcoming section provides an opportunity to observe and implement a project involving this pattern.

Interestingly, composite elements can also comprise other composites as child elements, tailoring to specific requirements. This pattern elegantly and flexibly allows the creation of unlimited hierarchical relationships between classes of the same type or between elements sharing the same interface. Besides, it potentially reduces the number of required classes. It paves the way to

construct intricate structures, provided a universally applicable pattern is defined for all the elements involved in the solution, across all levels. The composite pattern hence offers a streamlined solution for tackling complex object hierarchies.

Composite pattern implementation

The most effective way to comprehend any design pattern is to embed it within a program, thus experiencing its strengths and constraints firsthand. The rationale behind the employment of a design pattern should primarily be problem-oriented. The composite pattern should be harnessed only if the system necessitates implementing hierarchical relationships between classes sharing the same interface.

Consider a scenario where an online store offers individual products. Suppose the company decides to sell groups of these products as bundled offerings. Even though a bundled product is essentially a collection of individual products, it must also be considered a distinct product, with its own price, name, barcode, and other specifications.

As the system evolves, the hypothetical company might introduce more combinations of products as composite single items, thereby escalating the system's complexity. Maintaining such a system becomes increasingly challenging. The composite pattern aims to resolve this exact issue, offering the flexibility to embed an unlimited number of sub-elements within parent elements. This full-fledged implementation introduces new hurdles, such as database structure, performance, and entity relationships. However, the example scenario in this section zeroes in on the implementation part of the application, utilizing the C# language.

Given the scenario, and adhering to the architecture proposed by the composite pattern, our system would necessitate the following tasks:

- Establishing a common interface for the Product entity.
- Developing a concrete Product class that implements the Product interface.
- Defining the composite classes and their methods for adding and removing child elements.

- Specifying custom methods that are exclusive to the child class.

In line with the composite pattern definition and based on the requirements, an element may or may not possess child elements. Even though all elements share the same interface, a specific custom method must be defined in the composite class to differentiate single elements from composite ones. Based on the requirements specified in this section, our online store's implementation would mirror the structure represented in *Figure 11.3*:

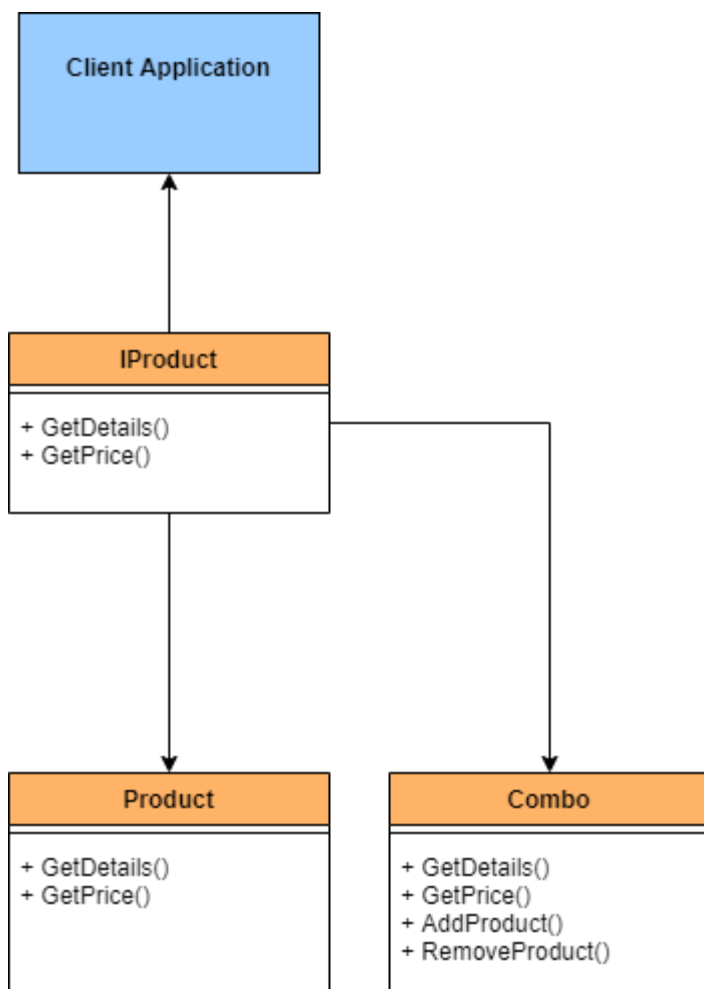


Figure 11.3: Composite pattern implementation

As illustrated in the preceding [Figure 11.3](#), both the `Combo` and `Product` classes share the same interface, though the `Combo` class exclusively houses methods for adding and removing products. Given that a `Combo` is a product encompassing other products, this class epitomizes the composite class as defined by the composite pattern. It's noteworthy that the apt use of interfaces enables us to craft valuable abstractions, as demonstrated through the implementation of various design patterns.

Deepening your understanding of advanced concepts such as interfaces, inheritance, and abstract classes is crucial. If these concepts seem foreign to you, it's recommended to refer to [Chapter 5](#), Introduction to Design Patterns, which elucidates these topics with detailed explanations and practical examples, employing the C# language.

The goal in this section is to construct an online store capable of creating products, with the provision that some products can represent a list of several products sold as a single entity. As previously articulated in this section, the first step towards implementation, in line with our objectives, is to create a common interface used by all products. A group of products in this context is referred to as a `Combo`. Thus, using Visual Studio, create a Console Application and formulate an interface that mirrors the structure displayed in the following [Figure 11.4](#):

```

1  namespace DesignPatternsBookChapter11.Models
2  {
3      0 references
4      public interface IProduct
5      {
6          0 references
7          int Id { get; set; }
8          0 references
9          string Name { get; set; }
10         0 references
11         decimal Price { get; set; }
12         0 references
13         string Category { get; set; }
14         0 references
15         void GetProductDetails();
16         0 references
17         decimal GetProductPrice();
18     }
19 }

```

Figure 11.4: IProduct interface

The interface should encapsulate all properties and methods to be shared between the **Product** and **Combo** classes. Each class will execute unique logic concerning details and pricing. It's important to note that not all products include sub-products. Moreover, only the composite classes should contain routines for adding and removing sub-elements. These methods are deliberately omitted from the interface to avert mistakes in the implementation and misuse of the composite pattern. The **Category** property, specified in the interface, will be used by the concrete class to differentiate between individual products and **Combos** for demonstration.

With the interface already established, the next phase involves creating the **Product** and **Combo** classes. In your Console application project, create a new class called **Product** and implement the **IProduct** interface, as depicted in the following [Figure 11.5](#):

```

1 namespace DesignPatternsBookChapter11.Models
2 {
3     0 references
4     public class Product: IProduct
5     {
6         1 reference
7         public int Id { get; set; }
8         2 references
9         public string Name { get; set; }
10        3 references
11        public decimal Price { get; set; }
12        2 references
13        public string Category { get; set; }
14
15        1 reference
16        public void GetProductDetails()
17        {
18            Console.WriteLine($"Name:{Name}. Price: {Price}. Category: {Category}");
19        }
20
21        1 reference
22        public decimal GetProductPrice()
23        {
24            return this.Price;
25        }
26    }
27 }

```

Figure 11.5: Product class

As evidenced in the code snippet, the class incorporates all properties and methods defined by the interface. The **GetProductDetails** method implementation exhibits a summary of the product details, such as **name**, **price**, and **category**. This serves purely demonstrative purposes. In real-world scenarios, this method would entail fetching information from a database or executing more complex logic to assemble product details. Nonetheless, the current implementation suffices to familiarize you with the composite pattern's behavior and to discern between a single **Product** and a composite one comprising child products.

The **GetPriceDetails** method in the **Product** class manifests the functionality to display the original price of the product. It currently does not accommodate the application of a special discount. Consequently, all objects having the individual product as part of their sub-product list (that is, **Combo** class) will inherit its value automatically. This structure enables us to maintain the independence between a single product and the composite objects, despite their shared interface. Ultimately, all composite objects are products themselves.

The subsequent phase necessitates the creation of the **Combo** class. A **Combo** is a product that houses associated products. Note that this differs from merely grouping objects. The **Combo** class retains the

same characteristics as an individual product, such as **name**, **price**, and other properties. Considering the diagram showcased in [Figure 11.2](#), a **Combo** is a product that encapsulates other nodes. Each individual node could include one sub-product, and recursively, a sub-product could be composed of several sub-products. This configuration presents a potent and flexible structure to depict a business process requiring a hierarchical relationship between products and sub-products. The **Combo** class implementation is revealed in the following [Figure 11.6](#):

```

1 reference
public class Combo : IProduct
{

    private List<IProduct> _subProducts;
    0 references
    public Combo()
    {
        _subProducts = new List<IProduct>();
    }

    #region PROPERTIES
    1 reference
    public int Id { get; set; }
    2 references
    public string Name { get; set; }
    5 references
    public decimal Price { get; set; }
    2 references
    public string Category { get; set; }
    #endregion
    2 references
    public void GetProductDetails()...

    2 references
    public decimal GetProductPrice()...
    0 references
    public void AddProduct(IProduct product)
    {
        _subProducts.Add(product);
    }
    0 references
    public void RemoveProduct(IProduct product)
    {
        _subProducts.Remove(product);
    }
}

```

Figure 11.6: Combo class

The **Combo** class, unlike the **Product** class, implements the **IProduct** interface and introduces a distinct difference. Within the **Combo** class, a private property has been added to represent the sub-products that form part of the combo.

Additionally, the class includes two specific methods: **AddProduct** and **RemoveProduct**. These methods are tasked with adding new

products to the sub-product list and removing existing ones, respectively. While the content of these methods remains concealed for demonstration purposes, the details of their implementation are provided further below.

The inclusion of a list of **IProducts** in the **Combo** class serves as an example of how to morph a regular individual class into a composite one. While the **Product** and **Combo** classes share similarities, the **Combo** class incorporates extra logic to manage the sub-products tied to a combo instance.

As depicted in [Figure 11.5](#), the **GetProductDetails** and **GetProductPrice** methods in the **Product** class are designed to solely focus on the product itself. In contrast, these same methods within the **Combo** class exhibit greater complexity, encompassing code to also oversee their sub-products. It is important to note that the details for each sub-product utilize methods derived from the child element, as can be seen in the subsequent [Figure 11.7](#):

```
2 references
public void GetProductDetails()
{
    Console.WriteLine($"Name:{Name}.  Category: {Category}");

    Console.WriteLine("****All the products in this combo****");

    foreach (var subProduct in _subProducts)
    {
        subProduct.GetProductDetails();
    }
}

2 references
public decimal GetProductPrice()
{
    this.Price = 0;

    foreach (var subProduct in _subProducts)
    {
        this.Price += subProduct.GetProductPrice();
    }

    Console.WriteLine($"***Total Price: {this.Price}***");
    return this.Price;
}
```

Figure 11.7: Combo class detail methods

The **GetProductDetails** method within the **Combo** class includes a

bespoke implementation that extends beyond merely displaying **Combo** details. It systematically executes a loop, traversing the sub-product list, and invoking the corresponding method from the child element. By doing so, it ensures the child class's implementation remains intact, while the parent class retains autonomy over its own methods.

It's crucial to recognize that a composite object could potentially be a child of other composite objects, thereby establishing an unlimited hierarchy of relationships among all objects sharing the common **IProduct** interface. Should the developed implementation fall short of allowing this complexity, it's indicative that the composite pattern has likely been implemented improperly.

Continuing the exploration of the **Combo** class methods, the **GetPriceDetails** method, as revealed in the preceding [Figure 11.7](#), iterates over each sub-product. During this process, it incrementally adds to the **Price** property of the **Combo** class to compute the final price. Consequently, the same method in the child class is differentiated from the composite one.

With this, our full composite pattern implementation is complete and ready for deployment. The stage is now set to utilize these classes within our client console application.

Consider a real-world scenario where an online store offers three products: a book, a laptop, and a monitor, each with their distinct pricing. Occasionally, a promotional offer allows the store to present a bundled package named **techCombo**, inclusive of all three products. Within the framework of the composite pattern, the individual items – the book, the laptop, and the monitor – would be defined using the **Product** class, while the **techCombo** would be constructed via the **Combo** class, reflecting the fact that the combo incorporates sub-products.

Once individual instances for the first three products are created, they must be added to the combo object, aligning with the composite pattern. This process is illustrated in the following [Figure 11.8](#):

```

0 references
public IActionResult Index()
{
    Product book = new Product() { Name = "Book", Price = 35.00m, Category = "Product" };
    Product laptop = new Product() { Name = "Laptop", Price = 1200.00m, Category = "Product" };
    Product monitor = new Product() { Name = "Monitor", Price = 150.00m, Category = "Product" };

    Combo comboOne = new Combo() { Name = "Combo One", Category = "Combo" };
    comboOne.AddProduct(book);
    comboOne.AddProduct(laptop);
    comboOne.AddProduct(monitor);

    Product keyboard = new Product() { Name = "Keyboard", Price = 20.00m, Category = "Product" };
    Product mouse = new Product() { Name = "Mouse", Price = 15.00m, Category = "Product" };
    Product mousePad = new Product() { Name = "Mouse Pad", Price = 25.00m, Category = "Product" };

    Combo comboTwo = new Combo() { Name = "Combo Two", Category = "Combo" };
    comboTwo.AddProduct(keyboard);
    comboTwo.AddProduct(mouse);
    comboTwo.AddProduct(mousePad);

    comboOne.AddProduct(comboTwo);

    comboOne.GetProductDetails();
    comboOne.GetProductPrice();

    return View();
}

```

Figure 11.8: The client application

Within the main method, the initial three lines are dedicated to the creation of the three individual products. Each product is defined by assigning specific values to the **Name**, **Price**, and **Category** properties. Following this, on Line 13, the **techCombo** object is instantiated.

Subsequent lines are engaged in adding the three products to the **techCombo** object. Given that the **Combo** class is equipped with specialized methods to manage the addition and removal of new sub-products, it becomes essential to familiarize yourself with the methods exclusive to the **Combo** class.

These methods, distinct to the **Combo** class and crucial to its functionality, are vividly highlighted in the [Figure 11.9](#) provided in the following:


```

3 references
public decimal GetProductPrice()...
7 references
public void AddProduct(IProduct product)
{
    _subProducts.Add(product);
}
0 references
public void RemoveProduct(IProduct product)
{
    _subProducts.Remove(product);
}

```

Figure 11.9: Addition (*AddProduct*) and removal (*RemoveProduct*) methods

Finally, the aim of this chapter has been to guide you in understanding and familiarizing yourself with the composite design pattern. This pattern has been a cornerstone in software development for many years and continues to be widely applied. I encourage you to embark on your own project, crafting a hypothetical scenario that allows you to translate the knowledge and insights gained from this chapter into new contexts and applications. By doing so, you'll have the opportunity to further explore and adapt this versatile pattern to suit various situations and challenges.

Conclusion

In this chapter, you've discovered that the composite pattern offers an elegant and straightforward solution to a complex problem, particularly when dealing with recursive hierarchical structures within an application. Renowned as one of the most widely used design patterns in software development, the composite pattern's implementation can indeed be intricate and demanding.

Effectively applying this pattern necessitates a thorough understanding of the specific business context and needs of the application. Abstracting the hierarchical dependencies between an element and its sub-elements can pose challenges, yet the composite pattern furnishes an elegant architectural framework that offers limitless scalability and extensibility.

Commonly deployed in systems responsible for orchestrating

workflows, comprising sequential steps and sub-steps, this pattern is also prevalent in libraries providing graphical hierarchical visual structures, such as diagramming tools.

Within the confines of the C# language, this chapter has equipped you with the skills to implement the composite pattern. You should now grasp how to leverage this pattern to craft simple and effective solutions for complex issues, aligning with the best practices of object-oriented programming.

As you advance to the next chapter, you will continue to deepen your knowledge of design patterns within the context of the C# language and the .NET platform. The upcoming content will introduce the proxy pattern, allowing you to integrate and apply the comprehensive insights and techniques you've acquired from the preceding chapters of this book.

Points to remember

- The composite pattern promotes uniformity by using the same interfaces across classes and subclasses, treating both individual objects and their compositions uniformly.
- In implementing the composite pattern, it is essential that the composite class houses a collection of its subclasses, allowing for the construction of complex structures.
- Remarkably flexible, the composite pattern can designate a composite class as a subclass within another composite structure. This recursive nature provides virtually unlimited abstraction and adaptability, fostering intricate hierarchical designs.

Multiple choice questions

1. In which of the following is the composite pattern suitable?
 - A. Systems that require non-hierarchical architecture.
 - B. Systems that contain a huge amount of types.
 - C. Any software based on the object-oriented programming paradigm.
 - D. Functionalities where elements must be represented

hierarchically.

2. Looking at the example given in this chapter, which alternative contains the composite class?

- A. IProduct interface
- B. Client application
- C. Combo class
- D. Product class

Answer

	D	
	C	

Questions

1. Explain the primary purpose of the composite pattern.
2. According to what you have learned in this chapter, create a hypothetical scenario in which the composite pattern could be applied, apart from the example given during this chapter.
3. Explain the type of scenarios in which the composite pattern can be used.
4. Are there any projects that you are currently working on that implement the composite pattern?

Key terms

- **Hierarchical structure:** The relationship between the parent and child objects.
- **Recursion:** A technique of making a function calls itself. This technique provides a way to break down the complicated problems =into simple problems, which are easier to solve.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Proxy Pattern with GRPC

Introduction

Implementing security and access control policies are essential aspects of most enterprise applications, as they are vital for the success of businesses in risk mitigation. There are design patterns that help build a robust architecture that encapsulates access to specific external resources and defines a simple standard that increases the code readability and maintainability.

The Proxy pattern helps us build a robust architecture that allows us to control access to a specific resource from an external source by allowing only specific operations to be used, filtering the data available for what client applications or consumers need to use. This pattern represents an elegant way to intercept requests to APIs and add extra functionality for client applications without changing the typical behavior of the source application.

Learning the Proxy pattern allows you to understand how to implement access control for simple and complex integrations between systems and libraries that share the same interface and

protocol but which data and operations need to be limited to the client application due to business and security requirements.

Within this chapter, following a step-by-step approach, you will be able to apply the Proxy pattern in practical examples using the C# language, and Blazor applications combined with GRPC protocol.

Structure

In this chapter, we will discuss the following topics:

- Proxy pattern definition
- Proxy pattern implementation
- Code samples using Blazor and GRPC

Objectives

After studying this chapter, you should be able to understand the theoretical and practical aspects of the Proxy pattern and identify the use of this pattern in .NET applications and legacy projects.

Proxy pattern definition

The primary goal of the Proxy pattern is to mediate the communication between two or more components to control access to the source functionalities in the context of integration.

The Proxy pattern, by definition, can easily be confused with the Adapter pattern. Both aim to mediate direct access to a specific resource, class, or functionality. However, despite their purposes seeming quite similar, the intention behind each is entirely different. The Adapter pattern provides an interface that is understandable to a client application if one of the classes or components involved has a different interface or output.

It is important to highlight that the Proxy pattern is not related to changing the client applications' output or data format but only to filtering and limiting access to certain functionality.

Proxy pattern types

Considering the Proxy Pattern can be used in different scenarios and

contexts, it is possible to split the pattern into the following sub-types:

- **Smart:** this type has the purpose of making different operations when integration is established between the client and the source application. This kind of Proxy is beneficial for making operations that are not part of the original functionality involved in the integration, such as logging operations.
- **Protection:** this Proxy type primarily aims to add security control and access limits to specific functionality. This type is commonly used in APIs where a profile or user from the client application should limit the methods available.
- **Cache:** in many scenarios in enterprise applications, the data returned by a method or API is not frequently changed. In this case, a proxy class can be implemented to intercept the requests and redirect the process to another process that would avoid unnecessary calls to expensive resources, such as transactional databases.
- **Virtual:** Sometimes, the Proxy pattern can simplify the output provided to a client application. Simplification in this context means reducing the methods or functions the source application exposes.
- **Remote:** this subtype is one of the most common uses of the Proxy pattern, which consists of creating a middleware responsible for intermediating access to a remote resource. The client application does not necessarily have to know where a certain resource is hosted, which means that a proxy component abstracts all the complexity behind the communication with the remote component.

One of the good practices presented in software development is the loose-coupling relationship between components. The Proxy pattern allows us to protect an original system from the requirements determined by other external systems. Therefore, if a client application needs to integrate with an existing third-party or external resource, all the parts involved in the integration can co-exist independently. This is accomplished using a mediator to preserve the source and target systems from interference regarding protocols and interfaces.

Proxy pattern structure

As part of the structural design patterns category, the Proxy pattern allows us to define workflows in the software architecture that would force the solution to implement restrictions to access objects and resources. Implementing this workflow makes it possible to intercept requests in an application to run extra custom routines based on functional and non-functional requirements.

Visually, the Proxy pattern representation has the structure represented in [Figure 12.1](#):

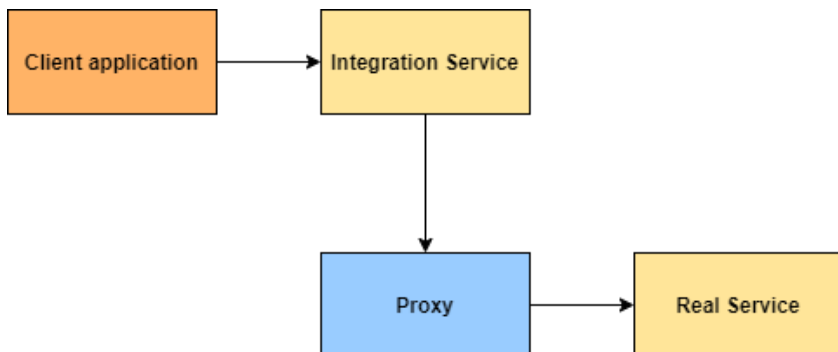


Figure 12.1: Proxy pattern diagram

[Figure 12.1](#) shows that a client application needs to access a specific service called **Real Service**. In this scenario, the request to the resource is intercepted by a **Proxy** layer, which allows us to apply extra operations such as access control and filtering. The **Proxy** layer can involve log operations for calls to the internal services and other custom routines.

According to the solution proposed by the Proxy pattern, a client application never accesses another service directly. This restriction can have different reasons for projects with different requirements, but the following are the most common justifications:

The **Client application** should not have access to all services provided by other applications.

Legal restrictions limit access to methods and features provided by other services.

The resource provided by an external service is significantly expensive in terms of costs and performance. This makes

intercepting the requests desirable and provides better alternatives to access the original resources. This scenario is commonly found when data caching returns information to a client application instead of serving the data from a traditional database source.

In the cases where the external resources should be controlled under different access rules by a client application, including policies that are not present in the source part of the integration from which the data is consumed. This is commonly found in integrations between systems developed by different companies that do not share the same requirements.

In situations where multiple client applications consume a resource, they have different requirements and expectations regarding features. The Proxy pattern can be used to redirect the request from a client application to the correct resource or feature without changing the implementation of the underlying code in both source and target projects.

When security policies are constantly changing by legal or business requirements involving a specific system consumed by many client applications.

In any integration between systems, full access to the source part of integration is usually limited to the scope established by contracts between companies. For instance, many large tech companies provide Artificial Intelligence services based on REST APIs. Access to the service may be limited by pricing tier, security policies, billing policies, or other underlying business rules.

Following recommended software architecture practices, any feature that is not directly related to the core of the business should be implemented in a structure apart. In the context of the Artificial Intelligence service, the original service should remain free of external interference, such as financial information and other business requirements beyond the core of the service, which is to provide the Artificial Intelligence service itself. In this scenario, the Artificial Intelligence service can be kept intact, and a Proxy layer can be developed to handle authentication, security policies, and other non-functional requirement rules.

Common use in real projects

The Proxy pattern is widely used in the market considering the extent of the capabilities that this pattern allows to implement. Solutions are made more robust by utilizing a pattern that minimizes changes in original classes and features because of requirements that are part of the application's core. Many APIs used in the industry involve certain levels of security and commonly implement the concepts stated by the Proxy pattern without many developers being aware that the pattern is actually used in many of those scenarios.

It is pretty common in solutions used by large-scale applications, which involve millions of users, to provide alternatives in handling requests based on the type of devices the users consume the applications. In this context, imagine a scenario in which a social media application has several users, and the type of device must customize the amount of data returned by the application. That means that certain types of devices may receive a more significant amount of data, and others a light version of the data because of limitations of the devices in terms of memory and processor capabilities. A proxy later can be used to intercept the requests from each device and apply the appropriate rules for each type of device before the response is returned from the server, as shown in [Figure 12.2](#):

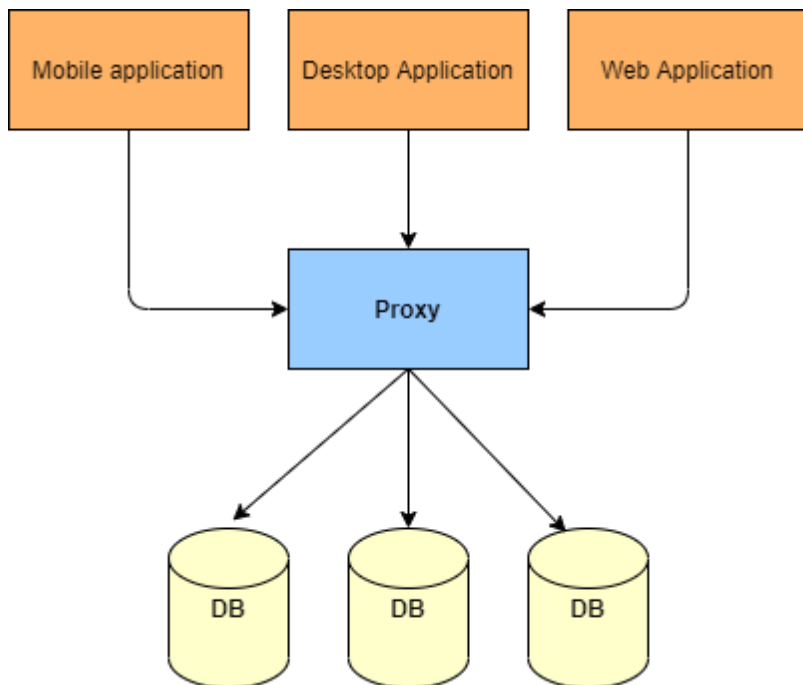


Figure 12.2: Proxy layer for social media

Initially, the information requested by the client applications is essentially the same, but the Proxy application applies custom policies for each kind of device, restricting the amount of data for mobile devices and returning the full content for desktop devices. Furthermore, the Proxy might be responsible for applying security checks, such as **JSON Web Token (JWT)** for mobile devices, and applying a different authentication method for Web and Desktop devices. All these aspects do not affect the source application.

In the next section, you can implement the Proxy design pattern for a hypothetical scenario close to what you would find in real scenarios. This application follows similar requirements that commonly exist in real-world applications to give you experience in implementing this pattern using the .NET platform with the C# language.

Proxy pattern implementation

This section contains specifications for a sample application that

handles requirements commonly found in real-world projects using the Proxy pattern. It is important to highlight that using any Design Pattern is not mandatory to solve a specific problem in software development. However, in most cases, their use represents an efficient way of providing a common, maintainable, and effective solution for common issues found in various projects across many industries. Therefore, the example explained in this chapter represents only one way of handling the given problem stated by the hypothetical scenario.

Scenario requirements

Imagine a scenario in which there is an application responsible for storing and managing information related to Online News. Each news story or article will have an advertisement associated.

The scenario has the following requirements:

- The news story or article has to be viewable on desktop, mobile, and web platforms.
- Each news story or article must have a category and can be associated with none, one, or multiple advertisements.
- Accessing the story or article from the Web platform should not allow users to read content from the Life Style category.
- Advertisements will not be included with the content if accessed by a mobile platform.

In the given scenario, the three distinct platforms (desktop, mobile, and web) will all consume the information from the same repository. The repository consists of a database that stores the content related to news, categories, and advertisements. However, to increase the complexity of the scenario, we can establish that other applications are consuming maintained by partner companies are consuming information from the same database. To access the database, a **ContentRepository** layer will be used. Therefore, as different sources and applications consume the same content, the **ContentRepository** layer should remain as it is, not being affected by requirements that are part of other applications or layers.

Practical example

To create this application, create a Console Application using Visual

Studio, then add the classes related to the example. The first class created is the **News** class, which contains the information relevant to a news story or article, as shown in the following [Figure 12.3](#):

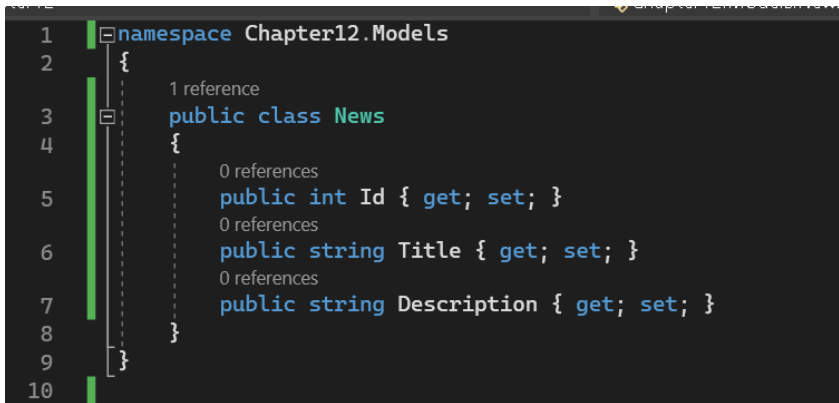


Figure 12.3: News class definition

The second class to be created is the **Advertisement** class, which will be a property of the **Content** class. The **Advertisement** class has the structure represented in [Figure 12.4](#):

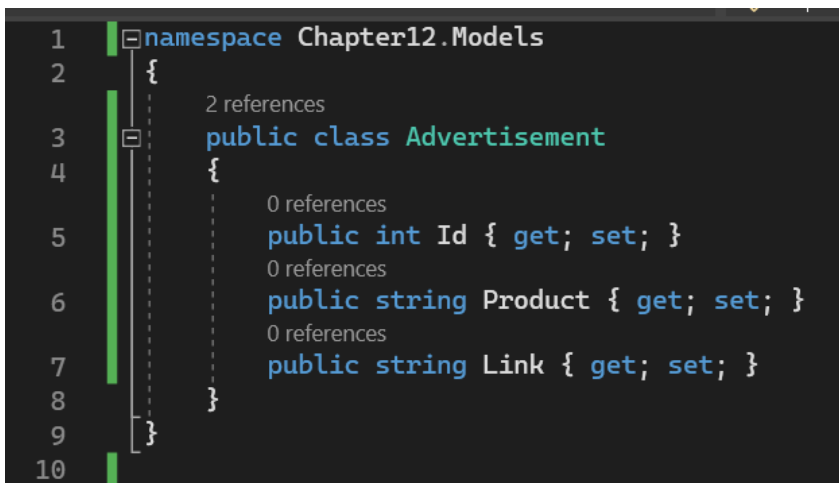
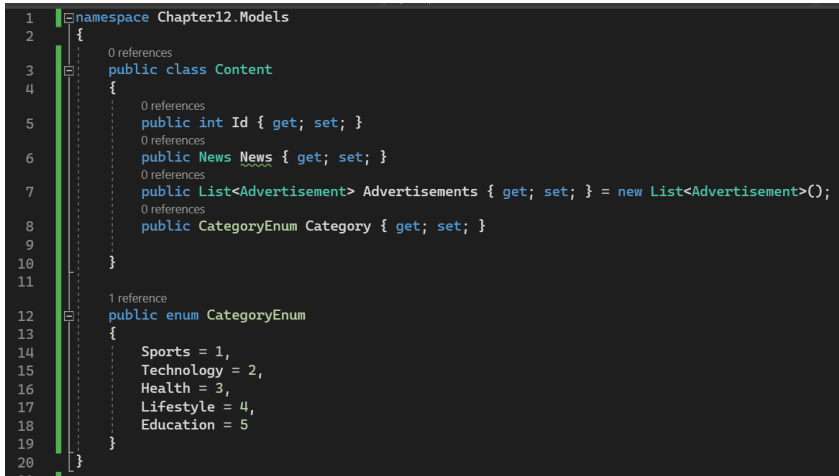


Figure 12.4: Advertisement class definition

The **Advertisement** class contains only the **Id**, **Product** name, and the hyperlink to redirect to the advertisement. As previously stated, the **Advertisement** class will be part of the **Content** class

and have a one-to-many relationship, considering the content (news story or article) may have one or multiple advertisements.

The **Content** class contains the news, a list of advertisements, and a category, as shown in [Figure 12.5](#):



```
1 namespace Chapter12.Models
2 {
3     0 references
4     public class Content
5     {
6         0 references
7         public int Id { get; set; }
8         0 references
9         public News News { get; set; }
10        0 references
11        public List<Advertisement> Advertisements { get; set; } = new List<Advertisement>();
12        0 references
13        public CategoryEnum Category { get; set; }
14    }
15
16    1 reference
17    public enum CategoryEnum
18    {
19        Sports = 1,
20        Technology = 2,
21        Health = 3,
22        Lifestyle = 4,
23        Education = 5
24    }
25 }
```

Figure 12.5: Content class structure

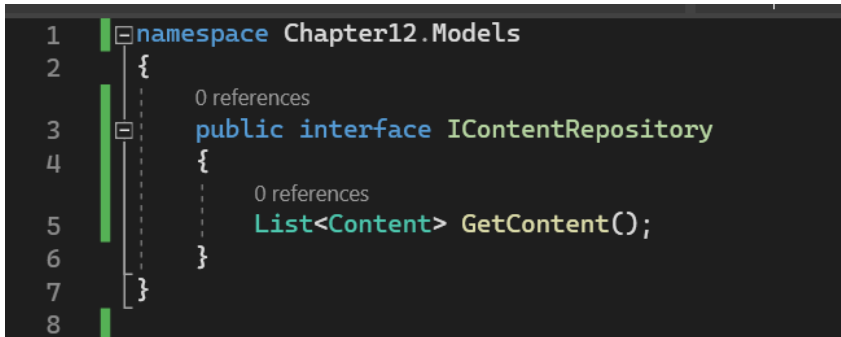
As shown in [Figure 12.5](#), the **Content** class file contains the **CategoryEnum** enum definition, which has five categories. At this point, all the necessary models are created, and it is now possible to implement the **Content** repository class, the underlying interface, and the Proxy pattern itself.

A layer called **ContentRepository** is responsible for establishing communication with the database and returning the necessary information to the consumer applications. For demonstration purposes in this section, the implementation of the **Content** repository is reduced to a simple **ContentRepository** class that simulates access to a database. Therefore, the hypothetical class will mock the content, news, and categories data.

In the Proxy pattern implementation, it is highly recommended that the **Proxy** class and the source and target classes in the integration share the same interface to keep the method names and properties with similar signatures, facilitating the integration between systems or layers. Therefore, the client application does not need to know the implementation from the existing repository as the access to the

resources does not happen directly but actually uses the Proxy layer as an intermediary. The Proxy will manage permissions, security filters, and other business requirements.

The next step in the implementation is to create the **IContentRepository** interface that will be shared by the repository and proxy class, with the implementation as shown in [Figure 12.6](#):

A screenshot of a code editor showing the definition of the **IContentRepository** interface. The code is in C# and is part of the **Chapter12.Models** namespace. The interface is named **IContentRepository** and contains a single method **GetContent()** that returns a **List<Content>**. The code is as follows:

```
1 namespace Chapter12.Models
2 {
3     0 references
4     public interface IContentRepository
5     {
6         0 references
7         List<Content> GetContent();
8     }
9 }
```

Figure 12.6: Content repository interface

The **IContentRepository** interface contains only one method for demonstration purposes. The **ContentRepository** and **Proxy** classes need to implement their own **GetContent** methods based on the requirements.

With the interface created, the next step is to create the **ContentRepository** class, which has the implementation presented in [Figure 12.7](#):

```

1 namespace Chapter12.Models
2 {
3     0 references
4     public class ContentRepository : IContentRepository
5     {
6         1 reference
7         public List<Content> GetContent()
8         {
9             List<Content> contentList = new List<Content>();
10            List<Advertisement> advertisements = new List<Advertisement>()
11            {
12                new Advertisement() { Product = "T-Shirt" },
13                new Advertisement() { Product = "Energy Drink"},
14                new Advertisement() { Product = "Game ticket"}
15            };
16
17            Content content1 = new Content();
18            content1.News = new News()
19            {
20                Title = "Football",
21                Description = "Sport news description"
22            };
23            content1.Category = CategoryEnum.Sports;
24            content1.Advertisements = advertisements;
25
26            Content content2 = new Content();
27            content2.News = new News()
28            {
29                Title = "Book",
30                Description = "Education news description"
31            };
32            content2.Category = CategoryEnum.Education;
33            content2.Advertisements = advertisements;
34
35            Content content3 = new Content();
36            content3.News = new News()
37            {
38                Title = "Lifestyle",
39                Description = "Lifestyle news description"
40            };
41            content3.Category = CategoryEnum.Lifestyle;
42            content3.Advertisements = advertisements;
43
44            contentList.Add(content1);
45            contentList.Add(content2);
46            contentList.Add(content3);
47
48            return contentList;
49        }
50    }
51 }

```

Figure 12.7: Content repository class

As shown in [Figure 12.7](#), the `GetContent` method simulates the database access and returns sample data that consists of a content list with three different elements with the following information:

- Content with the Sport category that has three advertisements
- Content with the Education category that has three advertisements

- Content with the Lifestyle category that has three advertisements

The given content is enough to simulate all the requirements defined at the beginning of this section for the three different platforms (mobile, desktop, and web). The last very step in the Proxy pattern implementation is creating a class responsible for intercepting the request from a client application. Based on the requirements, apply all the security, authentication, and data filter policies.

The purpose of the Proxy pattern is to provide an intermediate layer to manage anything that needs to be implemented in terms of security, policies, and filters that are not part of the original applications themselves. In the context of the scenario given in this section, the requests come from applications installed on different types of devices. A proxy class has been created to accept the requests and apply the requirements associated with each platform, avoiding the need to implement the rules in the repository directly.

Following good practices stated by the Proxy pattern implementation, the Proxy class must implement the same interfaces as the original class. Considering the given scenario, it means that the proxy class should implement the **IContentRepository** interface. The details of the **ContentRepositoryProxy** class need to implement all the custom requirements defined for each type of device and news category. Furthermore, the **ContentRepositoryProxy** class may contain rules regarding authentication, security policies, and any other custom implementation that goes beyond the functionality provided by the original source class or application. This means that the **ContentRepositoryProxy** class only includes logic that is related to the content filter.

Design Patterns, in general, are closely related to other practices in software development, such as the SOLID principles. They represent a combination of many good practices that are focused on writing high-standard code and providing efficient solutions for complex problems. As such, the best practices of the object-oriented programming paradigm are also followed by Design Patterns.

The **ContentRepositoryProxy** class implementation is presented

in the following [Figure 12.8](#):

```
1 namespace Chapter12.Models
2 {
3     5 references
4     public class ContentRepositoryProxy : IContentRepository
5     {
6         private DeviceType _deviceType;
7         ContentRepository contentRepository = new ContentRepository();
8
9         3 references
10        public ContentRepositoryProxy(DeviceType deviceType)
11        {
12            _deviceType = deviceType;
13
14            4 references
15            public List<Content> GetContent()
16            {
17                List<Content> contentList = contentRepository.GetContent();
18
19                switch (_deviceType)
20                {
21                    case DeviceType.Mobile:
22                        contentList.ForEach(x => { x.Advertisements = new List<Advertisement>(); });
23                        break;
24                    case DeviceType.Web:
25                        contentList = contentList.Where(x => x.Category != CategoryEnum.Lifestyle).ToList();
26                        break;
27                }
28
29                return contentList;
30            }
31
32            7 references
33            public enum DeviceType
34            {
35                Desktop = 1,
36                Mobile = 2,
37                Web = 3
38            }
39        }
40    }
```

Figure 12.8: Proxy class implementation

For clarification purposes, the **DeviceType** enum was created to explicitly handle the device types for the three platforms (**desktop**, **mobile**, and **web**). The class constructor receives a device type as a parameter, and the **GetContent** method has a statement to handle the different rules for each device type.

After bringing the data from the original repository, the **proxy** class filters the information for the mobile device type, removing all the advertisements. In the case where the request comes from the web platform, the Proxy class filters the content to exclude any content of the lifestyle category.

As shown in this example, the client application does not directly access the content from the repository, but it uses the **ContentRepositoryProxy** class instead, where rules are implemented in code as stated in the business requirements.

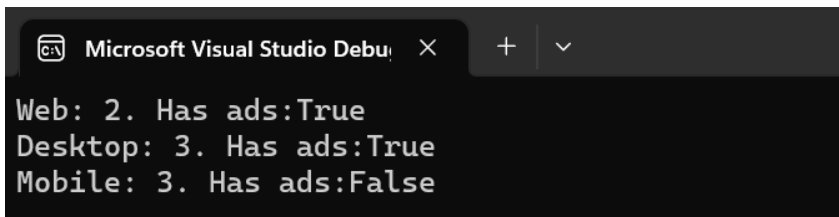
To see the results of the **Proxy** class implementation, you can call the proxy class in the Console Application passing distinct

parameters, as shown in [Figure 12.9](#):

```
7 static void Main(string[] args)
8 {
9     ContentRepositoryProxy contentRepositoryProxy =
10         new ContentRepositoryProxy(DeviceType.Web);
11
12     var contentListForWeb = contentRepositoryProxy.GetContent();
13
14     Console.WriteLine("Web: " + contentListForWeb.Count + ". Has ads:" +
15         contentListForWeb.Any(x=> x.Advertisements.Count > 0));
16
17     contentRepositoryProxy =
18         new ContentRepositoryProxy(DeviceType.Desktop);
19
20     var contentListForDesktop = contentRepositoryProxy.GetContent();
21
22     Console.WriteLine("Desktop: " + contentListForDesktop.Count + ". Has ads:" +
23         contentListForDesktop.Any(x => x.Advertisements.Count > 0));
24
25     contentRepositoryProxy =
26         new ContentRepositoryProxy(DeviceType.Mobile);
27
28     var contentListForMobile = contentRepositoryProxy.GetContent();
29
30     Console.WriteLine("Mobile: " + contentListForMobile.Count + ". Has ads:" +
31         contentListForMobile.Any(x => x.Advertisements.Count > 0));
32
33 }
```

Figure 12.9: Console application

In this simple program, the **Proxy** class is instantiated three times, one for each device type. The console displays the information about the number of contents returned by the proxy class and then checks if advertisements are associated with the content for each type. Considering the requirements in the given scenario, the mobile platform would receive the content of all categories but not advertisements. On the other hand, the web platform should receive advertisements from all categories except the lifestyle category. If you execute the Console application, you will get the results as shown in [Figure 12.10](#):



```
Microsoft Visual Studio Debug Console
Web: 2. Has ads:True
Desktop: 3. Has ads:True
Mobile: 3. Has ads:False
```

Figure 12.10: Proxy pattern results

The **Proxy** class works as expected and illustrates that the content repository class is not affected by any additional needs from the

business requirements. Additionally, any changes in the requirements would only be implemented within the **ContentRepositoryProxy** class, keeping the original content repository class safe from external interference.

gRPC with Blazor

gRPC is an open-source protocol developed by Google that allows us to use a high performance while using the Remote Procedure Call framework to establish communication between server and client using a protobuf contract. Considering this is an open-source protocol, many technologies, including .NET and C#, implement this.

Using gRPC with .NET projects, such as Blazor WebAssembly, can represent a great alternative to combine the simplicity of **Single Page Application (SPA)** development with a high-performance protocol to receive and send data between the Blazor WebAssembly application and backend services.

Usually, the integration between frontend and backend applications happens via REST APIs for most cases. Implementing another protocol, such as gRPC, may be challenging depending on the technologies used for the integration.

The Proxy pattern can be used to encapsulate the complexity of the implementation and calls to a gRPC service for Blazor applications. The service consumption looks similar to traditional REST APIs in terms of coding.

To demonstrate and verify how gRPC works, create a gRPC service using the default template in Visual Studio, as shown in [Figure 12.11](#):

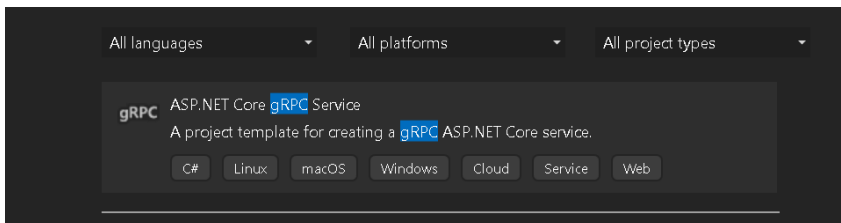


Figure 12.11: gRPC project template

Once created, the project should contain three main folders (**Properties**, **Protos**, and **Services**), as seen in [Figure 12.12](#):

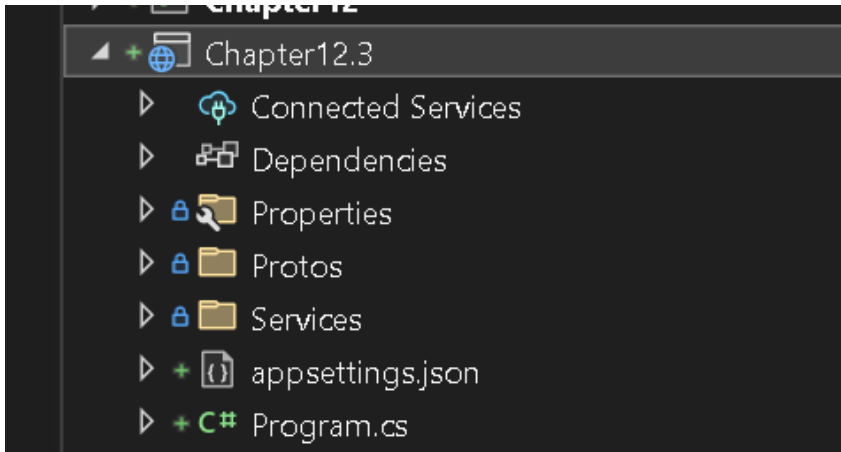


Figure 12.12: GRPC project structure

The **Services** contain the actual endpoints for the GRPC service, and the structure of the service is quite similar to a Controller for ASP.NET Core projects in general, as seen in [Figure 12.13](#):

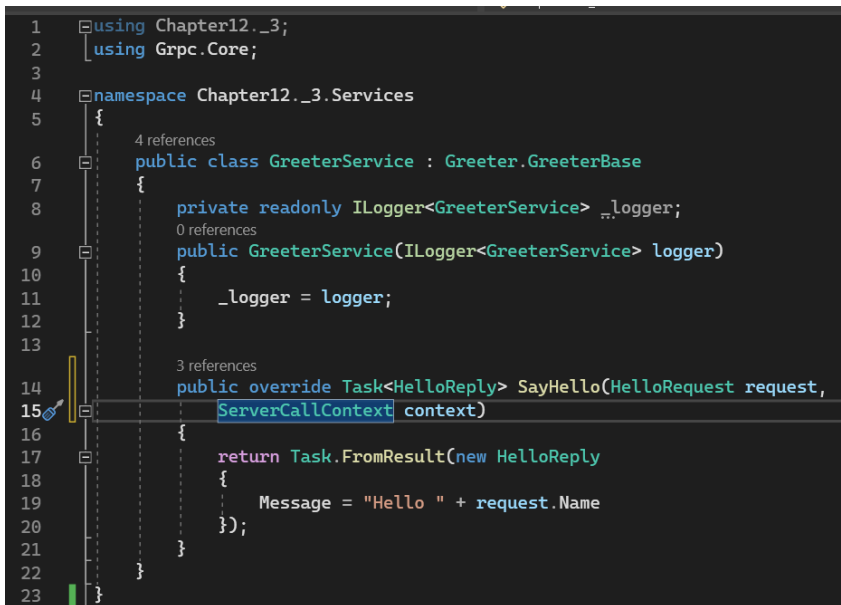
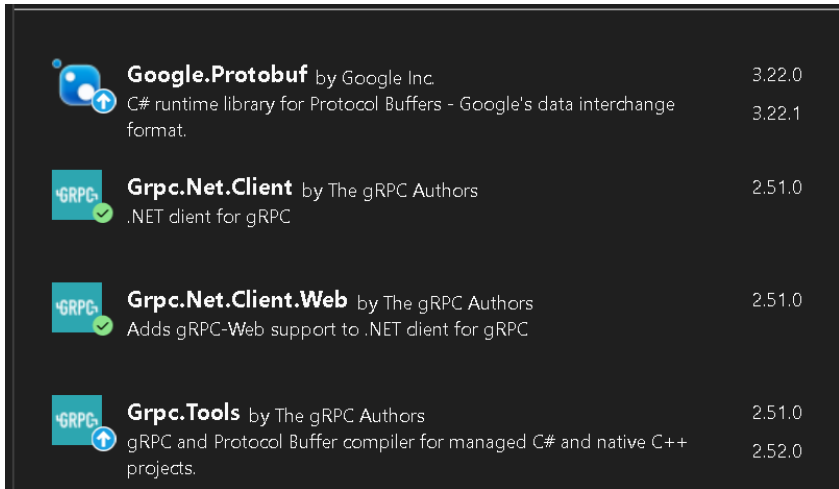


Figure 12.13: GRPC service code

The **SayHello** method, present in the default Visual Studio template for GRPC projects, receives a **HelloRequest** object that has a **name** property. The return of this method is a **HelloReply** object that returns a message that combines the word **Hello** with the name passed in the request.

To consume a GRPC service using a Blazor WebAssembly application, you can create a default template using Visual Studio and install the following packages shown in [Figure 12.14](#):



	Google.Protobuf by Google Inc. C# runtime library for Protocol Buffers - Google's data interchange format.	3.22.0 3.22.1
	Grpc.Net.Client by The gRPC Authors .NET client for gRPC	2.51.0
	Grpc.Net.Client.Web by The gRPC Authors Adds gRPC-Web support to .NET client for gRPC	2.51.0
	Grpc.Tools by The gRPC Authors gRPC and Protocol Buffer compiler for managed C# and native C++ projects.	2.51.0 2.52.0

Figure 12.14: Nuget Packages for GRPC

GRPC communication is meant to have a standard protocol between the client and server regarding requests and responses, including sharing the same data model on both sides. To facilitate the use of the same pattern, it is necessary to include the proto files in the client application as well, which can be done within a folder in the application, as seen in [Figure 12.15](#):

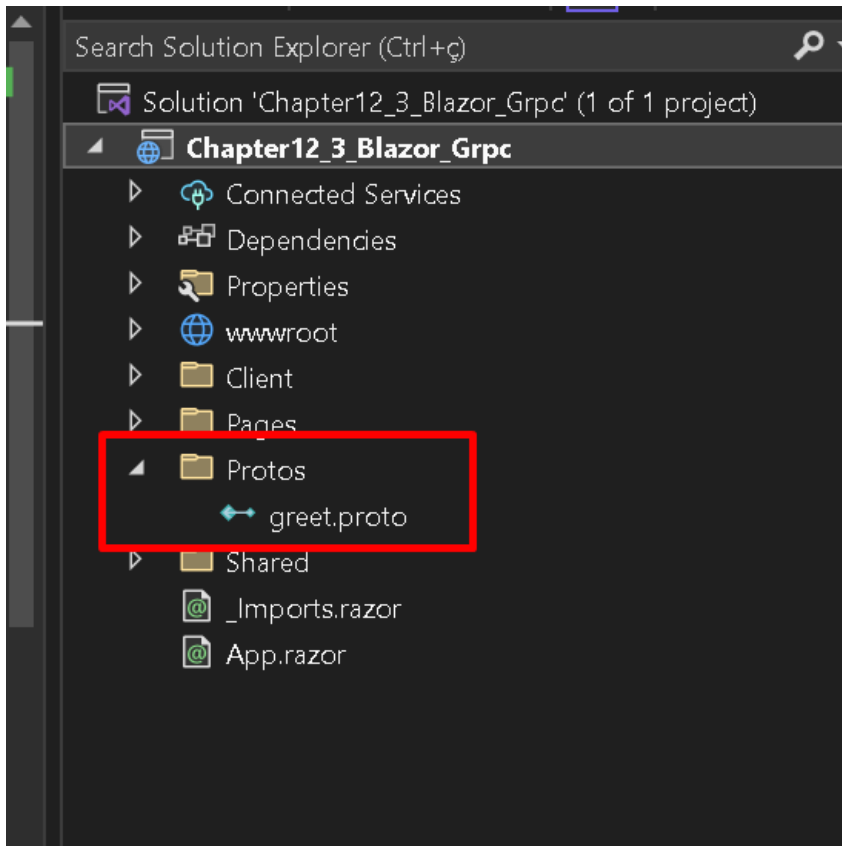


Figure 12.15: Proto files in the client application

It is important to highlight that you have to target the correct namespace for your application when placing the proto file in your client application, even if the rest of the content for the file is identical to the one used in the server application. Therefore, after copying the proto file from the server application to the client one, make sure you update the `csharp_namespace` property with the correct value with the name of your project, as seen in [Figure 12.16](#):

```

1  syntax = "proto3";
2
3  option csharp_namespace = "Chapter12_3_Blazor_Grpc";
4
5  package greet;
6
7  // The greeting service definition.
8  service Greeter {
9      // Sends a greeting
10     rpc SayHello (HelloRequest) returns (HelloReply);
11 }
12
13 // The request message containing the user's name.
14 message HelloRequest {
15     string name = 1;
16 }
17
18 // The response message containing the greetings.
19 message HelloReply {
20     string message = 1;
21 }

```

Figure 12.16: Namespace configuration for the proto file

After that, include the reference to the proto file in your **csproj** file, as seen in [Figure 12.17](#):

```

<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>

```

Figure 12.17: Proto file reference

Once all the configurations for the proto file and packages are made, you can start using the proxy classes created automatically by the project to encapsulate the communication between the client and server. To test the communication, you can adapt the **Index.razor** component to call the Greeter service, as seen in [Figure 12.18](#):


```

1  @page "/"
2  @using Grpc.Net.Client;
3  @using Grpc.Net.Client.Web;
4
5  <PageTitle>Index</PageTitle>
6
7  <h1>Grpc Response</h1>
8
9  <p>@helloReply.Message</p>
10
11  @code{
12      private HelloReply helloReply = new HelloReply();
13
14      protected override async Task OnInitializedAsync()
15      {
16          using
17          var channel = GrpcChannel.ForAddress("https://localhost:7272/",
18          new GrpcChannelOptions
19          {
20              HttpHandler = new GrpcWebHandler(new HttpClientHandler())
21          });
22          var client = new Greeter.GreeterClient(channel);
23          helloReply = await client.SayHelloAsync(new HelloRequest
24          {
25          });
26      }
27  }

```

Figure 12.18: Grpc client

As you can see in [Figure 12.18](#), the **HelloReply** and **HelloRequest** classes are used for the request and response, although we did not have to create these classes. They have generated automatically by parsing the proto file referenced in the *csproj* file.

GRPC may be a complicated protocol to understand and implement, however, all the complexity of the communication between the client and server is hidden by Proxy classes that are generated by Visual Studio tools when the project is built. This is a typical example of a good use of the Proxy Pattern.

Conclusion

As you have learned in this chapter, the Proxy pattern allows us to implement a simple solution for scenarios that require the use of distinct protocols to establish communication between two or more components, applying policies between them, and keeping all the custom changes required for any part of the integration in the Proxy layer.

This pattern is typical in cases where external applications, databases, and APIs, in general, need to be protected from

modifying their core implementation. The original source and target of the integration points remain intact.

With this chapter, you had the opportunity to learn how to implement the Proxy pattern in a real scenario using the C# language, and you learned how to apply the pattern using Blazor, GRPC, and Object-Oriented Programming paradigm.

In the next chapter, you will have the chance to continue your journey into Design Patterns using C# language and the .NET platform by learning the Command pattern and applying all the general knowledge gained in the preceding chapters of this book.

Points to remember

- The Proxy pattern helps us to intercept the access to functionalities of a source application applying business requirements without affecting the original implementation itself.
- A single solution can combine the Proxy pattern with the SOLID principle.
- It is possible to apply the Proxy pattern in any situation where access to certain functionality must be restricted due to security reasons.

Multiple choice questions

1. **In which scenario is the proxy pattern suitable?**
 - A. Integration between systems
 - B. Scenarios in which security is not a concern
 - C. Any project that uses the C# as language
 - D. None of them
2. **Looking at the example given in this chapter, which class contains the necessary implementation for the Proxy pattern?**
 - A. Content repository class
 - B. Client application
 - C. ContentRepositoryProxy class

Answers

	A	
	Q	

Questions

1. Explain the primary purpose of the Proxy pattern.
2. According to what you have learned in this chapter, create a hypothetical scenario in which the Proxy pattern could be applied apart from the example given in this chapter.
3. Explain the types of scenarios in which the Proxy pattern can be used.
4. Are there any projects you are currently working on implementing the Proxy pattern?

Key terms

- **Client application:** The layer of a system that usually contains a user interface and has a higher level in terms of abstraction.
- **System integration:** Communication between two or more components using defined protocols.
- **API:** Application programming interface usually made for allowing integration between different systems.
- **Cache:** In software development, it consists of storing data in a place that performs better than the original source.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

Command Pattern Using MediatR

Introduction

The Command Design Pattern is a versatile tool that consolidates all essential information related to a multifaceted operation into a single object. It efficiently handles complex tasks, including database transactions, API requests, and various integrations, thus simplifying the entire process for client applications. By encapsulating all the internal operations performed by the command class, it creates a cohesive structure that enhances both flexibility and control.

Being a preferred choice among database providers who need to integrate diverse programming languages, the Command Design Pattern is undeniably one of the most prevalent design patterns in software development today. Its widespread adoption across a vast array of projects worldwide attests to its effectiveness and relevance.

Learning the Command Pattern not only facilitates the identification of its application in legacy projects but also optimizes the time

spent in comprehending existing code. This can lead to significant resource savings and streamline the process of implementing new software, particularly when using the C# language and .NET platform.

In the forthcoming chapter, you will be guided through the practical application of the Command Pattern using the C# language. Together, we'll construct a realistic scenario step-by-step, providing a hands-on experience that reinforces your understanding of this valuable design principle. Whether you are a seasoned professional or new to this pattern, this chapter offers insights and techniques that can be invaluable to your software development journey.

Structure

In this chapter, we will discuss the following topics:

- Command pattern concept
- Examples in C# and .NET
- Use of the command pattern with MediatR

Objectives

After completing this unit, you will have gained the skills and understanding necessary to master the command pattern, a fundamental concept in modern software design. You'll be equipped to identify its use within .NET applications and existing projects, recognizing its implementations and variations. Beyond mere theoretical understanding, you'll also be able to apply the command pattern practically in real-world scenarios, employing its principles to create more robust, flexible, and maintainable code. Whether working on new developments or refining existing systems, this knowledge will serve as a valuable asset in your programming toolkit.

Command pattern definition

Design patterns play a vital role in simplifying solutions to complex problems by adhering to the best practices of object-oriented programming, software architecture, and SOLID principles. Although each pattern differs in its specific implementation and

ultimate objective, focusing on their underlying principles rather than their concrete implementation can render them more accessible and straightforward to grasp.

In this chapter, you will delve into the intricacies of the command design pattern, uncovering both its structure and primary purpose. Your journey will include a practical example, where you'll learn how to implement the pattern to integrate with an external financial service for an online store. Before embarking on this hands-on experience, it is crucial to familiarize yourself with the basic structure of the command pattern and understand its core intention.

The command pattern serves to shield a client application (the requester) from the complexity of an operation performed by another layer of the application (the receiver). Since the receiver is solely responsible for executing actions, the number of sub-actions required by a request becomes irrelevant. This pattern's central philosophy is to enable the execution of multifaceted tasks and amalgamate the results into a unified high-level operation invoked by the client application. This not only facilitates the execution of actions but also the reversal of those actions, akin to transactions commonly used in database operations and extensive business processes.

The command pattern's high-level abstraction can be visualized in [Figure 13.1](#), illustrating the interaction between the client application and the command pattern:

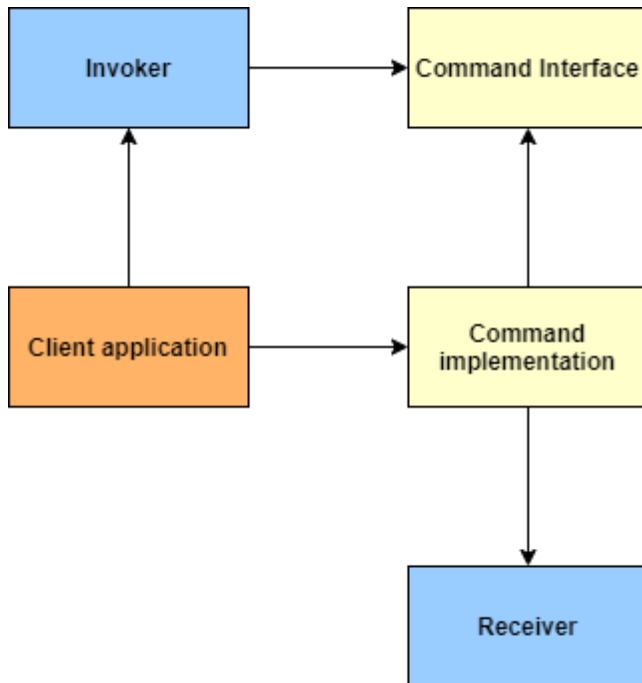


Figure 13.1: Command pattern diagram

The [Figure 13.1](#) also delineates the distinct responsibilities assigned to each layer, offering a clear and comprehensive view of how the components interconnect and function together. Whether a seasoned developer or a novice, this chapter aims to deepen your understanding of one of the most versatile design patterns, enhancing your ability to create elegant and robust software.

Implementing the command pattern necessitates several distinct components: the client application, invoker, command interface, command implementation, and receiver. Initially, this configuration typically represents the standard implementation of this design pattern across various scenarios. However, more complex situations requiring the execution of multiple commands may call for slightly different implementations.

Utilizing a common interface for all commands within the same context not only ensures consistency but also enhances the testability of the application. By injecting a command into a class, it's possible to simulate the behavior of an actual, concrete command class, thus enabling more robust testing and development.

Given that the command pattern's definition may seem abstract at times, it's highly recommended to deepen your understanding by engaging with a real-world application of the pattern. To that end, following the practical tutorial accompanying this chapter is essential. This hands-on experience will allow you to see the command pattern in action and apply its principles in a manner directly relevant to real-world software development, cementing your grasp of this powerful design concept.

Proxy pattern implementation

Given that the command pattern's primary objective is to abstract the execution of intricate operations from a client application, gaining authentic experience with this pattern necessitates the creation of an application designed to tackle the very problem the command pattern seeks to resolve. As previously noted in the opening section of this chapter, and detailed in the pages that follow, you will have the opportunity to embark on a hands-on project. This project involves constructing an abstraction for a financial transaction, a process commonly encountered in online stores. It represents a real-world application of the command pattern, and the requirements for this practical exercise are as follows:

- The online store must facilitate payment through credit card transactions, providing a seamless and secure experience for the user.
- Upon the completion of a purchase, the system must automatically update the inventory by reducing the quantity available of the purchased product, ensuring accurate tracking and availability.
- If a product supplied by a third-party vendor is sold, the system must promptly notify the relevant suppliers, fostering clear communication and timely replenishment.

In the event of a failure in any of the above operations, the system must be capable of rolling back all the associated actions, preserving data integrity and minimizing the potential for errors. This ensures that the entire transactional process is coherent and reliable, even in the face of unforeseen challenges.

In the given scenario, the system's construction necessitates the

development of a common command interface to be implemented across all operations. Although these operations may be entirely distinct from one another, their specific implementation must remain hidden from the client application. Naturally, executing this example to its full extent would encompass interactions with external APIs, database integration, and coordination with third-party systems. This includes communication with suppliers, though it's worth noting that the underlying execution will be simulated for the purpose of this exercise, keeping the focus squarely on the command pattern's implementation.

As for the business model classes required, the example mandates the development of models that encapsulate various facets of the purchasing process. This includes key data related to the purchase itself, as well as interconnected details concerning the customer, product, and supplier. These entities must be carefully defined and structured to reflect the operational needs of an online store and to align with the principles of the command pattern. A visual representation of these relationships and their integration within the system can be found in [Figure 13.2](#), providing a clear and comprehensive overview of how they interact within the context of this specific design pattern:

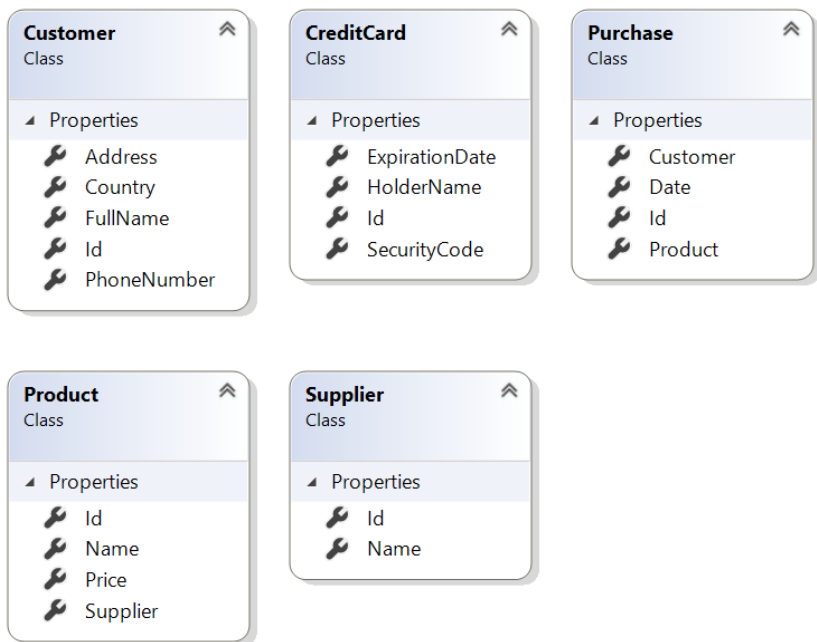


Figure 13.2: Command Pattern class diagram

The **Purchase** class serves as the central hub within this system, given its vital role in the command pattern implementation. It is designed to act as a parameter, encapsulating all the essential information required to fulfill the operations outlined in the requirements. By coherently organizing this data, the **Purchase** class streamlines the execution of the various tasks and enhances the maintainability of the code. The specific implementation of this class, complete with its attributes and methods, is illustrated in the subsequent [Figure 13.3](#), providing a clear visual guide to its structure and functionality within the broader system:

```

namespace DesignPatternsBookChapter13
{
    6 references
    public class Purchase
    {
        0 references
        public int Id { get; set; }
        0 references
        public DateTime Date { get; set; }
        0 references
        public Product Product { get; set; }
        0 references
        public Customer Customer { get; set; }
        0 references
        public CreditCard CreditCard { get; set; }
    }
}

```

Figure 13.3: Purchase class

Certainly! Here's an improved version of the text:

In the context of the relationships between the **Purchase** class and the entities of **Product** and **Customer**, the complete realization of this hypothetical scenario necessitates the subsequent development of the **Customer** class. This component plays a critical role in defining and managing user-specific details within the system. A visual depiction of the implementation of the **Customer** class, including its key attributes and methods, is provided in [Figure 13.4](#). This illustration serves as a guide to understanding its structure, importance, and interaction with other elements within the broader framework:

```

namespace DesignPatternsBookChapter13
{
    1 reference
    public class Customer
    {
        0 references
        public int Id { get; set; }
        0 references
        public string FullName { get; set; }
        0 references
        public string PhoneNumber { get; set; }
        0 references
        public string Address { get; set; }
        0 references
        public string Country { get; set; }
    }
}

```

Figure 13.4: Customer class

The **Product** class maintains a one-to-one relationship with the **Supplier** class, reflecting the direct association between a product and its provider. For the sake of this demonstration, the **Supplier** class focuses solely on the properties that reference the model, omitting the specific methods responsible for handling the integration process with the suppliers. This streamlined approach serves to conceal the underlying complexity of supplier integration, allowing the example to concentrate on the essential structure and relationships without delving into the intricate details of communication and coordination with supplier systems. It ensures that the focus remains on illustrating the fundamental concepts of the command pattern and the connections between the classes within this context.

Extra classes

In accordance with the requirements outlined in the provided scenario, the implementation of both the **Product** and **Supplier** classes is integral to the example explored in this chapter. These classes, each fulfilling distinct roles within the system, have been carefully constructed to reflect the specific needs and constraints of the hypothetical online store. A detailed visual representation of their structure, attributes, and relationships is provided in the

subsequent [Figure 13.5](#), serving as a clear and insightful guide to understanding their function within the broader context of the command pattern implementation:

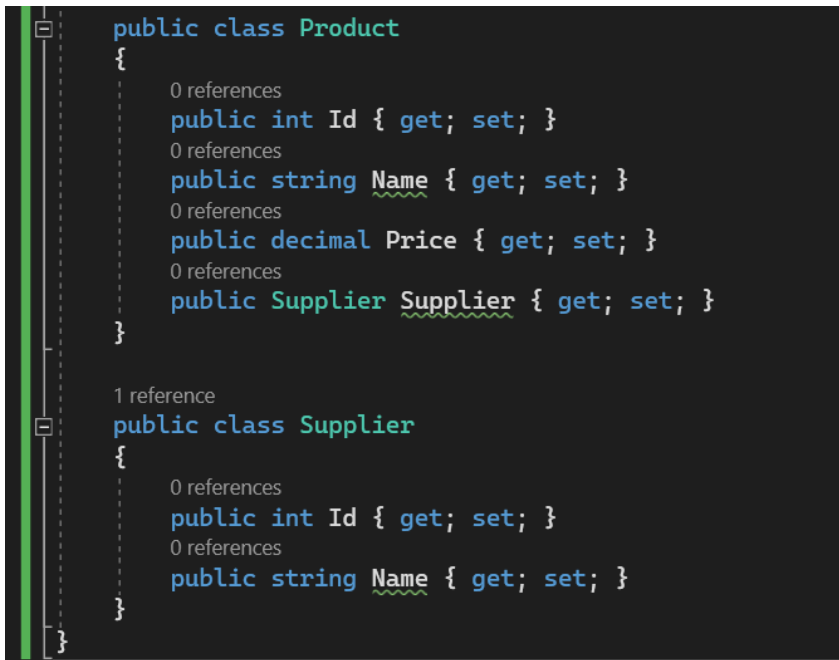
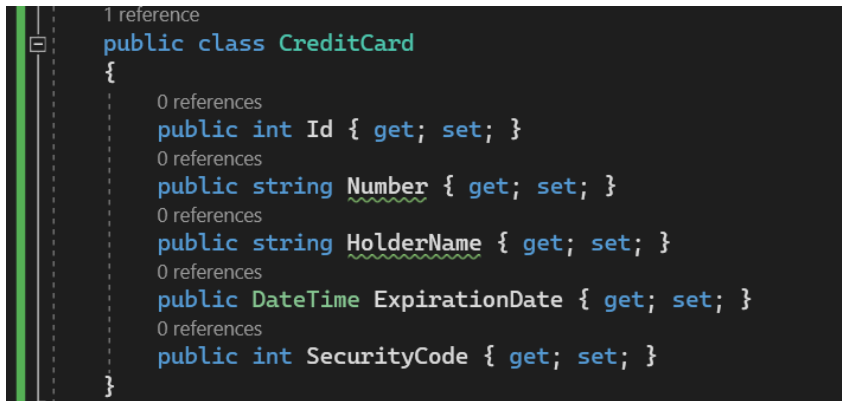


Figure 13.5: Product and supplier classes

Next, we have the **CreditCard** class, which is intricately connected to the **Purchase** class within the system. This class is responsible for handling the credit card information required for payment processing and forms a vital component of the transactional workflow. A detailed illustration of the **CreditCard** class's implementation, highlighting its attributes, methods, and relationship with the **Purchase** class, can be found in the following [Figure 13.6](#). It provides an insightful view into the role and structure of this essential element within the broader context of the command pattern example.



```
1 reference
public class CreditCard
{
    0 references
    public int Id { get; set; }
    0 references
    public string Number { get; set; }
    0 references
    public string HolderName { get; set; }
    0 references
    public DateTime ExpirationDate { get; set; }
    0 references
    public int SecurityCode { get; set; }
}
```

Figure 13.6: Credit card class

At this juncture, we have successfully crafted all the requisite classes pertaining to the models needed to realize the example set forth in this chapter. Within this framework, the command pattern will take on the role of orchestrating diverse operations, all of which adhere to a common interface. Central to this design, the constructor will encompass a purchase object, encapsulating the details needed for each transaction. Building on this foundation, the ensuing step involves the creation of the interface, which adheres to the same structural principles. A visual illustration of this interface, displaying its alignment with the overall design and the way it integrates with the other components, is provided in the following [Figure 13.7](#). This visualization offers a clear insight into how the interface functions within the broader context of the command pattern and sets the stage for the subsequent phases of implementation:

```

namespace DesignPatternsBookChapter13
{
    12 references
    public interface IPurchaseCommand
    {
        6 references
        void ExecuteOperation();
        7 references
        void RevertOperation();
    }
}

```

Figure 13.7: Common interface

The specified interface necessitates the implementation of two distinct methods: one tasked with executing a custom operation, and the other dedicated to reverting that operation should any exceptions arise. In alignment with the scenario described in this section, three separate command classes must be constructed, each corresponding to a specific operation: processing credit card payment, managing storage of product information, and executing supplier notification.

It's worth noting that for this illustrative example, it is not essential to implement the actual actions within these command classes. Instead, to effectively demonstrate the behavior of the implementation, you may simply use the `Console.WriteLine` method to emulate the underlying functionality.

To commence this process, the first command class will showcase the credit card operation. This component will serve as the backbone for handling payment transactions within the system, translating the abstract concepts of the command pattern into tangible code. A visual representation of this class's structure and implementation is provided in the following [Figure 13.8](#), offering a concise and informative view of its role within the broader framework of the design:


```

public class CreditCardPaymentCommand : IPurchaseCommand
{
    private readonly Purchase _purchase;

    0 references
    public CreditCardPaymentCommand(Purchase purchase)
    {
        this._purchase = purchase;
    }

    3 references
    public void ExecuteOperation()
    {
        Console.WriteLine("Integration with the credit card provider");
    }

    3 references
    public void RevertOperation()
    {
        Console.WriteLine("Something failed in the payment");
    }
}

```

Figure 13.8: Payment command class

In [Figure 13.8](#), it's evident that the class in question implements the **IPurchaseCommand** interface, taking a purchase object within its constructor. This approach forms the required pattern that all command classes must adhere to in the given example.

Considering the complete purchase process, it is comprised of three distinct operations. Notably, the implementation for two of these key functions—supplier notification and storage product management—is missing. Consequently, the subsequent task that needs to be addressed is the creation of the functionality responsible for reducing the product quantity information within the storage. This is an integral component of the entire process.

The underlying command class governing this operation contains the necessary implementation, which is meticulously illustrated in [Figure 13.9](#):

```

public class StorageManagerCommand : IPurchaseCommand
{
    private readonly Purchase _purchase;

    0 references
    public StorageManagerCommand(Purchase purchase)
    {
        this._purchase = purchase;
    }

    3 references
    public void ExecuteOperation()
    {
        Console.WriteLine("Operation to decrease the product quantity availability");
    }

    3 references
    public void RevertOperation()
    {
        Console.WriteLine("Something failed in the storage operation");
    }
}

```

Figure 13.9: *Storage command implementation*

In [Figure 13.9](#), you will notice that the storage command class adheres to the same pattern as the payment command. The difference lies solely in the implementation of the two methods, which are modified to execute distinct operations corresponding to each of them. Employing the same interface is a strategic architectural decision; it ensures that all the command classes throughout the system are aligned with a revert process as stipulated by the initial requirements of this section.

With this understanding, only one command class remains to be implemented: the one pertaining to supplier notification. Given that each supplier may necessitate a unique integration approach, the command design pattern can be thoughtfully combined with other patterns. One such example is the strategy pattern, elaborated in [Chapter 15, Observer Pattern](#), of this book.

Once the complexity of the supplier command class has been distilled, the finalized implementation is detailed in [Figure 13.10](#). This illustration underscores the integration of principles from previous sections to build a coherent and efficient design tailored to the specific needs of supplier notification:

```

public class SupplierCommand : IPurchaseCommand
{
    private readonly Purchase _purchase;

    0 references
    public SupplierCommand(Purchase purchase)
    {
        this._purchase = purchase;
    }

    3 references
    public void ExecuteOperation()
    {
        Console.WriteLine("Operation to notify the supplier");
    }

    3 references
    public void RevertOperation()
    {
        Console.WriteLine("Something failed in the supplier notification");
    }
}

```

Figure 13.10: Supplier command implementation

With all three essential command classes now established, attention turns to the construction of the class entrusted with encapsulating the operations conducted by these command classes. This is a pivotal part of the system's design, closely aligned with the requirements set forth at the outset.

Should the execution of an operation either throw an exception or yield an unexpected result, the system is mandated to initiate the corresponding revert process for that specific operation. This necessitates a robust architecture where the managing class for the command classes must unequivocally ensure that the behavior aligns with expectations.

Figure 13.11 provides a detailed visual representation of this design, illustrating how the class directly managing the command classes functions to guarantee the anticipated behavior. It offers a clear view of the interplay between different elements, reinforcing the strategic importance of the class within the overall system architecture:

```

public class PurchaseLogic
{
    private readonly IPurchaseCommand _purchaseCommand;
    0 references
    public PurchaseLogic(IPurchaseCommand purchaseCommand)
    {
        this._purchaseCommand = purchaseCommand;
    }
    0 references
    public void ConfirmPurchase()
    {
        try
        {
            this._purchaseCommand.ExecuteOperation();
        }
        catch (Exception)
        {
            this._purchaseCommand.RevertOperation();
        }
    }
}

```

Figure 13.11: Purchase logic class

The **PurchaseLogic** class is designed to receive a purchase command class that complies with the purchase command interface, incorporating a specific method to oversee the execution of the operation defined by the concrete command class. As depicted in the code illustrated in [Figure 13.11](#), the **ConfirmPurchase** method includes a try-catch block, specifically engineered to manage any failures that might arise during the purchase execution process.

This design strategy ensures that the client application relies on the **PurchaseLogic** class to orchestrate operations, rather than invoking the underlying methods directly from the command class. Such an approach not only consolidates the control but also guarantees that the revert operation is consistently invoked if something unexpected occurs.

With all the requisite classes now in place to leverage the Command design pattern infrastructure, you have the flexibility to implement these sample classes within various contexts, such as a Console application or any other project that aligns with your needs for executing the operations embedded in each class.

In this particular chapter, a Console application serves as the platform to demonstrate the creation of multiple instances of command classes, as well as the instantiation of model classes. This encompasses information related to customers, suppliers, products,

and purchases. The first step involves crafting instances for customer and credit card objects, infusing them with hypothetical data. A visual representation of this process is conveniently presented in [Figure 13.12](#), offering a clear and concise guide to the task at hand:

```
Customer customer = new Customer()
{
    Id = 1,
    FullName = "Customer example"
};

CreditCard creditCard = new CreditCard()
{
    Id = 1,
    ExpirationDate = new DateTime(2030, 12, 31),
    HolderName = "Customer example",
    Number = "1111222233334444",
    SecurityCode = 123
};
```

Figure 13.12: Customer and credit card objects

With the foundational objects successfully established, the subsequent stage involves the generation of a new instance of the purchase object. This process requires careful association with the freshly created customer and credit card entities. [Figure 13.13](#) visually illustrates this critical step, providing a clear representation of the relationships and associations that have been formed within this key component of the process:

```

Purchase purchase = new Purchase()
{
    CreditCard = creditCard,
    Date = DateTime.UtcNow,
    Customer = customer,
    Product = new Product()
    {
        Name = "Book",
        Supplier = new Supplier
        {
            Name = "Book Store"
        }
    }
};

```

Figure 13.13: Purchase object

In this example, the purchase object is comprised of a single product, linked to a specific supplier. In a real-world application, such information would likely be sourced from a database or an external API. However, for the sake of this demonstration, the instances of these objects are constructed manually.

Each command class anticipates receiving a purchase object within its constructor. As a result, the identical instance of the purchase object will be disseminated across each type of command class instance, encompassing payment, storage management, and supplier notification. It's essential to recognize that a comprehensive purchase process, in the context presented, necessitates the coordination of operations across all three command classes.

In the client application, the procedure calls for the creation of an instance for each distinct command type. These instances must then be passed as parameters to the **PurchaseLogic** class instances.

[Figure 13.14](#) provides a visual guide to this intricate arrangement, illustrating the exact method by which the components interact and interrelate within the broader system architecture:

```
IPurchaseCommand paymentCommand = new CreditCardPaymentCommand(purchase);
IPurchaseCommand storageCommand = new StorageManagerCommand(purchase);
IPurchaseCommand supplierCommand = new SupplierCommand(purchase);

new PurchaseLogic(paymentCommand).ConfirmPurchase();
new PurchaseLogic(storageCommand).ConfirmPurchase();
new PurchaseLogic(supplierCommand).ConfirmPurchase();
```

Figure 13.14: Client application implementation

Take note that in this system, the purchase class object bears the responsibility for invoking the **ConfirmPurchase** method. This method incorporates a try-catch block specifically designed to manage the underlying execution of the purchase operation. In the given example, even though all three requisite operations are executed, the implementation is flawed. This deficiency lies in the absence of a routine that unequivocally ensures the reversal of all operations, should any one of the three fail.

It's important to stress that the existing implementation would not be incorrect if there were no specific business requirement mandating the reversion of all three operations as part of a single transaction in the event of a singular failure. However, given that such a requirement is indeed present, the implementation must undergo some adjustments to fully satisfy this need.

The necessary modifications aim to affirm that the system meets the stipulated conditions, with the requirement being addressed through the integration of the transaction class. A detailed illustration of how this class functions within the broader context of the system is provided in [Figure 13.15](#), offering a clear and insightful view of the adaptations that ensure compliance with the particular demands of this operation:

```

1 reference
public class TransactionPurchaseCommand : IPurchaseCommand
{
    private readonly List<IPurchaseCommand> _purchaseCommands;
    private List<IPurchaseCommand> executedPurchaseCommands;
    private List<IPurchaseCommand> failedPurchaseCommands;

    0 references
    public TransactionPurchaseCommand(List<IPurchaseCommand> purchaseCommands)
    {
        this._purchaseCommands = purchaseCommands;
        executedPurchaseCommands = new List<IPurchaseCommand>();
        failedPurchaseCommands = new List<IPurchaseCommand>();
    }

    3 references
    public void ExecuteOperation()
    {
        foreach (var purchaseCommand in _purchaseCommands)
        {
            try
            {
                purchaseCommand.ExecuteOperation();
                executedPurchaseCommands.Add(purchaseCommand);
            }
            catch (Exception)
            {
                failedPurchaseCommands.Add(purchaseCommand);
            }
        }

        if (failedPurchaseCommands.Any())
        {
            RevertOperation();
        }
    }

    4 references
    public void RevertOperation()
    {
        foreach (var purchaseCommand in executedPurchaseCommands)
        {
            purchaseCommand.RevertOperation();
        }
    }
}

```

Figure 13.15: Transaction purchase command class

Certainly! Here's an improved version of the text:

The transaction purchase command class is constructed to receive a list of purchase commands and is designed to implement the same purchase command interface, adhering to a consistent pattern. This specific implementation aligns with the requirement that necessitates the reversal of all operations if even a single one fails. Although this approach might appear more intricate than the previous implementation, it more accurately encapsulates and represents the underlying requirements.

Within the client application, it becomes essential to instantiate the

transaction command class. This step supersedes the creation of the three individual instances of the purchase logic class that were present in the earlier design. [Figure 13.16](#) provides a vivid depiction of this new structure, illustrating how the creation of a unified transaction command class streamlines the process, ensuring the fulfillment of the specified conditions and requirements:

```
IPurchaseCommand paymentCommand = new CreditCardPaymentCommand(purchase);
IPurchaseCommand storageCommand = new StorageManagerCommand(purchase);
IPurchaseCommand supplierCommand = new SupplierCommand(purchase);

List<IPurchaseCommand> purchaseCommands = new List<IPurchaseCommand>()
{
    paymentCommand,
    storageCommand,
    supplierCommand
};

new TransactionPurchaseCommand(purchaseCommands).ExecuteOperation();
```

Figure 13.16: Client application with a transaction

Utilizing the Command design pattern, we have the ability to craft code that is both readable and transparent in expressing the underlying intentions of the requirements. Within the context of this chapter's scenario, the requirements sought to ensure that all necessary operations related to an online store purchase would be executed cohesively. Should one or more of these operations fail, the entire purchase process must be reverted, treating all individual operations as components of a single unified process.

This design pattern proves particularly advantageous in this setting, offering a structured and clear method for handling complex operations. It is my hope that this example not only illuminated your understanding of the Command pattern but also equipped you with the discernment to recognize and apply this pattern in legacy projects. The nuanced implementation provided here serves as a practical guide, fostering both comprehension and capability in navigating the intricacies of this versatile pattern.

MediatR and Command pattern

MediatR is a library used in C# that facilitates the implementation of the mediator pattern, enabling objects to communicate without knowing each other's identities. The Command pattern turns a

request into a stand-alone object that can be passed and saved like any other object. This pattern separates the object that sends a request from the one that actually executes it. Together, MediatR can be used to send Command objects to their respective handlers, allowing them to work in unison.

In a C# application utilizing MediatR, Commands can be defined as requests that implement the **IRequest** interface. Handlers are then created to deal with these Commands. By encapsulating request handling logic inside Command handlers and using MediatR to dispatch these Commands, the architecture becomes more modular and maintainable. MediatR handles the routing, letting the application adhere to the Single Responsibility Principle, as each Command handler only needs to handle one specific type of command.

The integration of MediatR and the Command pattern simplifies the management of complex applications with many interacting components. It fosters loose coupling since components don't need to know about each other, only the Commands they send or receive. This leads to a more maintainable codebase and enables easier testing by isolating functionality into testable units. Such an architectural choice can be particularly beneficial in applications following the CQRS pattern, where the separation of reading and writing operations aligns well with the usage of Commands and MediatR.

Conclusion

Throughout this chapter, you have witnessed the vital role that the Command pattern plays, especially in scenarios that require executing multiple operations as a cohesive group. It is adept at abstracting the complexity of these operations from the client application. This pattern finds frequent use in .NET applications, as most database providers harness the Command pattern to simplify the execution of database operations within applications developed using the C# language. The insights you've gained from this chapter empower you to craft your own command structure from the ground up, applying it in real-world projects. You can also seamlessly blend this pattern with others aimed at encapsulating the execution of operations, further isolating the client application.

In exploring this chapter, you have not only learned how to practically implement the Command pattern using the C# language but also how to devise simple solutions to intricate problems. This entails the application of best practices in object-oriented programming, enriching your understanding and technical proficiency. Your newfound ability to harness the Command pattern will enable you to create more modular and maintainable code.

Looking ahead to the next chapter, your exploration of design patterns using the C# language and the .NET platform will continue to flourish. You'll delve into the Strategy pattern, building upon the foundational knowledge acquired in the earlier chapters of this book. This journey will further refine your skills, equipping you to meet the demands of modern software development with increased confidence and expertise.

Points to remember

- The Command pattern effectively conceals the complexity behind the execution of operations from the client application. By adhering to a consistent interface and pattern, it ensures a unified approach to handling various tasks.
- The versatility of the Command pattern allows it to be combined seamlessly with other design patterns, such as the Strategy pattern. This enables more robust and flexible solutions that can be tailored to specific needs.
- In scenarios where multiple operations need to be managed as a single transaction, the Command pattern is highly recommended. Its structured approach simplifies the orchestration of complex processes, promoting efficiency and reliability.

Multiple choice questions

1. In which scenario is the Command pattern recommended?
 - A. For simple operations
 - B. When the integration between systems is complex
 - C. When operations need to be executed and handled as a transaction

D. None of them

2. Looking at the example given during this chapter, which class contains the necessary implementation for the proxy pattern?

A. Purchase logic

B. Customer

C. Purchase

D. Purchase command

Answers

	C	
	D	

Questions

1. Explain the primary purpose of the Command pattern.
2. According to what you have learned in this chapter, create a hypothetical scenario where the Command pattern could be applied apart from the example given during this chapter.
3. Explain the in which type of scenarios the Command pattern can be used.
4. Are there any projects that you are currently working on that implement the Command pattern?

Key terms

- **Transaction:** In software development, a transaction represents a group of tasks that need to be executed together, and a row back process is executed for all the tasks in case of at least of them fail.
- **Receiver:** In the context of the Command design pattern implementation, a receiver is the source class responsible for executing a certain operation.
- **Invoker:** In the context of the Command design pattern implementation, an invoker is an intermediate class or layer

responsible for calling a command class.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 14

Strategy Pattern Using Azure C# and Azure Functions

Introduction

In the multifaceted realm of software development, numerous solutions can be applied to a specific problem, and different approaches may yield the same result. Design patterns, however, offer a unique advantage in this landscape. They serve as a unified toolbox, providing proven solutions to recurrent issues encountered across diverse companies and a wide array of project types.

Within this framework, the Strategy pattern emerges as an elegant and efficient tool. It's designed to simplify the complexity of algorithms implemented across multiple classes that share a similar goal. This pattern's uniform approach makes the code more adaptable and easy to maintain, ensuring a seamless integration across various parts of a system.

By mastering the Strategy pattern, you open doors to recognizing its application in legacy projects built on the C# language and .NET

platform. It doesn't merely provide a historical perspective; it equips you with a high-level abstraction tool for managing diverse operations in both new and existing projects. This understanding can significantly enhance the quality of your code, making it more extensible and robust.

This chapter offers you a valuable opportunity to delve into the practical application of the Strategy pattern. By following a step-by-step guide using the C# language, you'll build a real-world scenario, gaining hands-on experience. This tangible connection between theory and practice will enrich your knowledge and prepare you for the challenges of modern software development.

Structure

In this chapter, we will discuss the following topics:

- Strategy pattern concept
- Examples in C# and .NET
- Practical usage of Azure Functions for strategy pattern

Objectives

After studying this unit, you will gain a comprehensive understanding of the Strategy pattern, a vital concept that transcends mere theoretical knowledge. Not only will you grasp its fundamental principles, but you'll also become adept at identifying its applications, particularly within .NET applications and existing projects. Recognizing the Strategy pattern in existing code is a critical skill, enabling you to analyze how others have approached similar challenges and glean insights from their solutions.

Beyond mere comprehension, this unit equips you with the practical skills needed to apply the Strategy pattern in real-world scenarios. By bridging the gap between theory and practice, you'll be able to utilize this pattern effectively in your projects, enhancing the maintainability, flexibility, and overall quality of your software. This hands-on experience with the Strategy pattern will serve as a robust foundation, aiding you in crafting elegant solutions to complex problems.

Strategy pattern definition

Understanding the difference between runtime and compile time in the C# language is essential before diving into the concrete definition of the strategy pattern. Compile time refers to the phase when operations are identified and recognized while the application is being built, and before the project runs. For example, in a Web application using ASP.NET Core, any issues like syntax errors in the C# implementation will result in build errors in the Visual Studio IDE. All types and classes identified by the compiler must be valid, which means they must be known at compile time, including their specific implementation in the context of the C# language.

On the other hand, runtime refers to behaviors that are only identifiable when the application is actively running. Issues related to interfaces and logical implementations, dependent on specific conditions like page requests or database connections, are detected at this stage. Essentially, compile time deals with known types, classes, and syntactic behaviors, while runtime handles more uncertain, variable behaviors that depend on certain factors.

In this context, the strategy pattern plays a significant role, allowing the type of a method or implementation to be altered at runtime. Depending on specific conditions, the code may create different classes in various scenarios, offering flexibility in encapsulating the complexity of operations. Similar principles are found in other patterns, such as the factory method.

The strategy pattern specifically involves specifying a common interface for classes with similar purposes. A strategy pattern class then determines which concrete class should be instantiated based on parameters, conditions, or custom verification, aligning with both business and technical requirements. This approach fosters a more adaptable and maintainable code structure, emphasizing the importance of the strategy pattern in modern software development.

Strategy pattern structure

In this chapter, you will delve into the intricate details of the strategy design pattern. Along the way, you'll discover how to apply this concept to a real-world scenario, specifically implementing the business requirements for a global delivery application. But before

embarking on that journey, it's vital to acquaint yourself with the underlying structure of the strategy pattern and comprehend its primary objective.

At the heart of the strategy pattern is a context class that orchestrates the decision-making process, determining which concrete class needs to be instantiated. During runtime, this context class defines the specific strategy class that will be engaged by the primary routine, a relationship that is visually depicted in [Figure 14.1](#). This dynamic approach allows for adaptable and modular code, aligning the technical implementation with the ever-changing needs of the business environment:

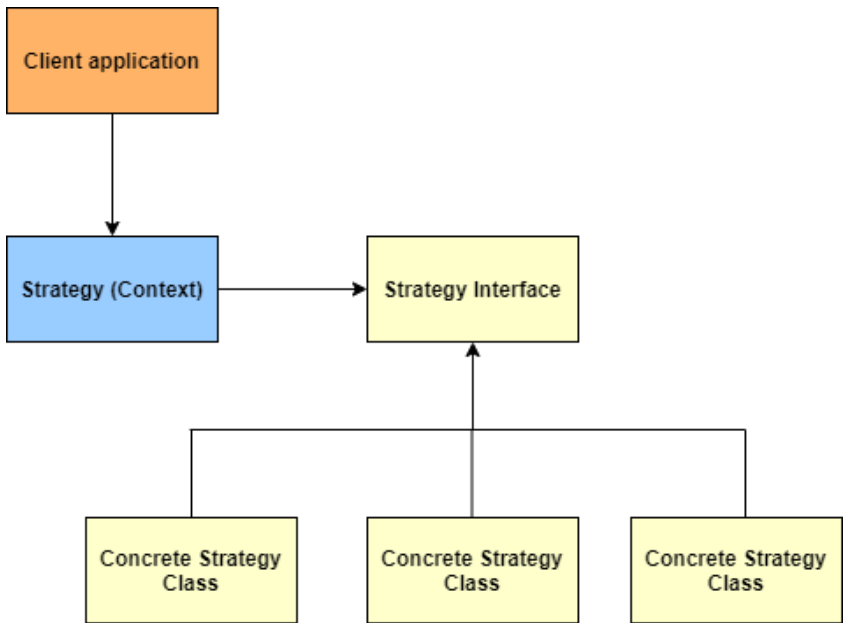


Figure 14.1: Strategy pattern diagram

As depicted in [Figure 14.1](#), all the concrete classes conform to the same interface, facilitating a unified approach. The logic behind creating the correct object is abstracted from the client application, as a specific routine may have multiple options to select from. The client application doesn't need to know which class is required; instead, the entire strategy is orchestrated by the context class, hiding the complexity of the decision-making process.

In real-world applications, this approach can be particularly powerful. For example, a complex project involving hundreds of classes and intricate micro-operations may require a carefully designed strategy pattern. Abstracting their complexity from the client application, the creation of the context class represents the most significant challenge in implementing this pattern. Within this chapter, you will have the hands-on opportunity to construct the strategy pattern from the ground up. Guided by the requirements of a hypothetical yet realistic scenario, you'll gain practical insights into how this design approach can be applied to solve complex problems, enhancing the adaptability and maintainability of your code.

Strategy pattern implementation

In software development, the application of design patterns should be carefully aligned with the specific problems they aim to solve. Different problems necessitate unique solutions, and diverse requirements call for tailored implementations that cater to all necessary aspects without compromising the principles of object-oriented programming. With this in mind, consider a hypothetical scenario: a global delivery company looking to regulate processes per city, where each locale has its own rules concerning taxes, revenue, employment, and pricing. For example:

- New York City imposes a 10% tax on all transactions and allocates 25% of profits for delivery.
- Los Angeles levies a 20% tax on transactions and earmarks 10% of profits for delivery.
- São Paulo stipulates a 10% tax, 15% profit margin, and mandates permanent employees.
- Any city without specific regulations defaults to New York's requirements.

This scenario presents a multitude of implementation avenues, each with its own challenges and nuances. A crucial consideration is that such applications may need to scale across many cities, each with varying business rules. Designing a single application to support the myriad legal requirements would be formidable, given the differing complexities of legal regulations among cities.

To maintain architectural consistency and handle this complexity, the strategy pattern can be a valuable tool. By employing this pattern, the client application's object instantiation—which might involve intricate implementation—can be kept simple, especially at the application's higher levels. An intelligently designed strategy infrastructure would take on the responsibility of creating the correct instance for each city, seamlessly loading all the local regulations for taxes, employment, and profitability.

The foundational elements for this example would be the classes related to user and delivery entities. Since this chapter focuses on the implementation of the strategy pattern, these models have been simplified, omitting some properties and methods that might exist in a real-world application. The first class to be crafted is the `User` class, the details of which are illustrated in the subsequent [Figure 14.2](#). This step marks the beginning of a hands-on journey into applying a powerful design pattern to a complex, globally scalable problem:

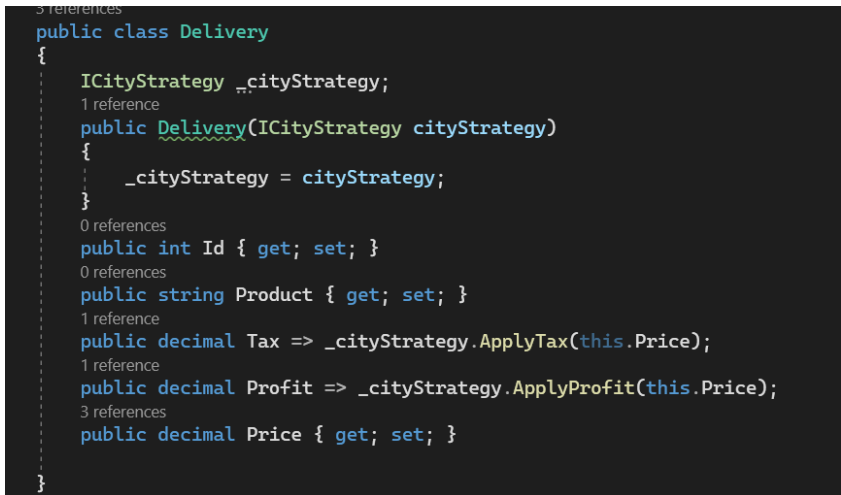
```
2 references
public class User
{
    0 references
    public int Id { get; set; }
    0 references
    public string FullName { get; set; }
    0 references
    public DateTime BirthDay { get; set; }
    0 references
    public string Address { get; set; }
    1 reference
    public string City { get; set; }
    2 references
    public string CityCode { get; set; }
}
```

Figure 14.2: User class

The `User` class is designed to encapsulate only the essential information required for the strategy pattern class, particularly including a property for the city code. Within the context of this example, each city's unique requirements necessitate the creation of

a distinct strategy object, reflecting the varying implementations.

Complementing the **User** class, a **Delivery** class must also be constructed. This class captures the foundational information concerning the delivery entity, encapsulating the basic details necessary for the application's functionality. The structure of this class, as well as its relationship to the overarching strategy, can be viewed in the following [Figure 14.3](#), illustrating the streamlined yet adaptable design that caters to the complexities of the scenario:



```
3 references
public class Delivery
{
    ICityStrategy _cityStrategy;
    1 reference
    public Delivery(ICityStrategy cityStrategy)
    {
        _cityStrategy = cityStrategy;
    }
    0 references
    public int Id { get; set; }
    0 references
    public string Product { get; set; }
    1 reference
    public decimal Tax => _cityStrategy.ApplyTax(this.Price);
    1 reference
    public decimal Profit => _cityStrategy.ApplyProfit(this.Price);
    3 references
    public decimal Price { get; set; }
}
```

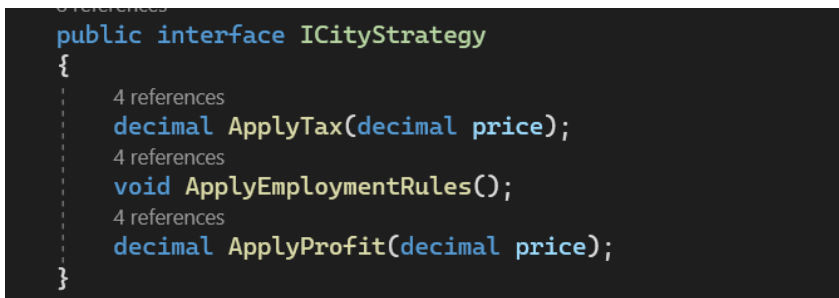
Figure 14.3: Delivery class

In accordance with the stipulated requirements, each city operates under distinct rules related to tax and profit. These unique regulations imply that the underlying properties of the **Delivery** class must be dynamically altered, leveraging the capabilities of the strategy class. Consequently, upon defining the basic structure of the class, as depicted in the preceding screenshot, its implementation must be further refined. Specifically, it must be adapted to accommodate the injection of specialized logic. This adaptation enables the class to accurately modify the tax and profit values in response to the unique rules governing each city, offering a flexible and context-sensitive solution that tailors itself to the diverse regulatory landscape.

Creating the city strategy class

To enhance both the testability and extensibility of the strategy pattern implementation, it's prudent to opt for interfaces instead of abstract classes when defining the base structures. This choice aligns with modern development practices and provides a more flexible and maintainable foundation.

Building on this critical consideration, the next component that requires creation is the city strategy class. This class acts as a linchpin in the system, stipulating the contract that governs the application of various city-specific requirements. It encompasses methods dedicated to handling the nuanced rules associated with tax, profit, and employment in each distinct locale, integrating them into a cohesive and robust strategy. A visual representation of this class, highlighting its essential features and the relationships it fosters within the system, is provided in the following [Figure 14.4](#):

A screenshot of a code editor showing the definition of the `ICityStrategy` interface. The code is written in C# and includes three methods: `ApplyTax`, `ApplyEmploymentRules`, and `ApplyProfit`. Each method signature is preceded by a comment indicating the number of references it has: "4 references". The interface is enclosed in curly braces, and the code is color-coded with green for keywords and blue for identifiers and literals.

```
public interface ICityStrategy
{
    4 references
    decimal ApplyTax(decimal price);
    4 references
    void ApplyEmploymentRules();
    4 references
    decimal ApplyProfit(decimal price);
}
```

Figure 14.4: City strategy interface

The `City` strategy interface is notable for its exclusive focus on methods rather than properties. This is in line with its specific role, which centers on handling the custom implementation tailored to the unique requirements of each city. Since the class's responsibility does not extend to maintaining the correct state of delivery or user objects, properties are unnecessary.

Considering that different objects need to be created for each city, it is more efficient to designate a concrete class for each city, each implementing the city strategy interface. This avoids the confusion and complexity that could arise from attempting to encapsulate multifaceted business logic within a single, unwieldy class. Another beneficial practice is to append the pattern name as a suffix to the classes involved in the design pattern implementation. This

facilitates pattern recognition among other developers working on the project and can streamline analysis, particularly in the context of legacy projects.

With the **City** strategy interface now in place, the next step involves modifying the previously specified **Delivery** class. This entails injecting the interface into the constructor method of the **Delivery** class, a critical step that integrates the interface's functionality into the class's behavior.

Additionally, the get methods for the tax and profit properties are defined by the underlying methods specified within the interface. This structure ensures a consistent and cohesive relationship between the interface and the class, with the interface effectively dictating the behavior of key properties. A visual illustration of these relationships and modifications is provided in the following [Figure 14.5](#), offering a clear and detailed view of how these components interact within the system:

```
public class Delivery
{
    ICityStrategy _cityStrategy;
    1 reference
    public Delivery(ICityStrategy cityStrategy)
    {
        _cityStrategy = cityStrategy;
    }
    0 references
    public int Id { get; set; }
    0 references
    public string Product { get; set; }
    1 reference
    public decimal Tax => _cityStrategy.ApplyTax(this.Price);
    1 reference
    public decimal Profit => _cityStrategy.ApplyProfit(this.Price);
    3 references
    public decimal Price { get; set; }
}
```

Figure 14.5: Delivery class with the strategy interface

The constructor of the **Delivery** class is designed to be dynamic, aligning with the strategy design pattern's principles. Therefore, it doesn't receive a concrete city strategy class; instead, the class to be instantiated is determined at runtime. This means that the specific class type remains unknown until the execution of the routine that

contains the strategy implementation. Given the infrastructure for concrete strategy classes, the next step is to create a distinct class for each city, with each class implementing the city strategy interface.

However, this approach could become challenging if the client application were suddenly expanded to a large number of different cities. This growth would necessitate the creation of many individual classes, following the pattern of one class per city. Managing this complexity would be difficult, requiring frequent application redeployments to accommodate new cities. In such scenarios, an alternative approach might be more suitable, such as a single class whose states are controlled by a comprehensive database structure. This structure would correctly support the specific rules for each city, streamlining maintenance and scalability.

For the current example, which outlines requirements for three cities (New York, Los Angeles, and Sao Paulo), three corresponding classes need to be created. The creation process could start with the New York City class, the structure of which is illustrated in the following *Figure 14.6*. This approach ensures that each city's unique requirements are accurately represented, laying the foundation for effective strategy pattern implementation:

```
2 references
public class NewYorkStrategy : ICityStrategy
{
    2 references
    public void ApplyEmploymentRules()
    {
        Console.WriteLine("No permanent employees");
    }

    2 references
    public decimal ApplyProfit(decimal price)
    {
        return price * 0.25m;
    }

    2 references
    public decimal ApplyTax(decimal price)
    {
        return price * 0.10m;
    }
}
```

Figure 14.6: New York strategy class

The **NewYorkStrategy** class, adhering to the **ICityStrategy** interface, is responsible for determining the employment rules for New York City. It is included as an input to the constructor of the **Delivery** class, encapsulating the logic for applying tax and profit calculations to the delivery price. The methods governing these calculations are found in the underlying class that implements the **ICityStrategy** interface, following the specific requirements outlined earlier in this section.

In line with the approach established for New York City, two additional classes must be created: one for Sao Paulo and another for Los Angeles. Both of these classes will also implement the **ICityStrategy** interface, which will be injected into the **Delivery** class based on the user's city. Though developing individual classes for each city may initially seem like a monumental task, given the potential number of cities to be implemented, this complexity serves a purpose.

The primary goal of the strategy pattern is to simplify the client application by abstracting complexity into the lower layers of the software. This means constructing a more intricate infrastructure in the underlying parts of the application, promoting flexibility and maintainability.

With the first concrete class for New York already established, the next step involves specifying similar classes for Sao Paulo and Los Angeles, each implementing the same interface. As an example, the following details present the implementation for the **LosAngelesStrategy** class, which adheres to the same structural principles as its counterparts, as seen in [Figure 14.7](#):


```

1 reference
public class LosAngelesStrategy : ICityStrategy
{
    2 references
    public void ApplyEmploymentRules()
    {
        Console.WriteLine("No permanent employees");
    }

    2 references
    public decimal ApplyProfit(decimal price)
    {
        return price * 0.10m;
    }

    2 references
    public decimal ApplyTax(decimal price)
    {
        return price * 0.20m;
    }
}

```

Figure 14.7: Los Angeles strategy class

As depicted in the preceding [Figure 14.7](#), the class name carries the **strategy** suffix, easing the identification of the strategy pattern for future maintenance. At first glance, the implementation of this class might seem redundant. However, in real-world scenarios, determining profit and tax values could require complex operations such as database calls or integration with external services.

To preserve the integrity of the design, it is advisable to keep the implementation isolated from external changes, like unexpected parameters that could affect the values produced by the tax and profit methods. This approach aligns well with good software development practices, reflecting the principles stated in the SOLID guidelines. By adhering to these principles, any change applicable to one city only necessitates modifications within the corresponding strategy class.

Emphasizing the importance of the **Single Responsibility Principle (SRP)**, it's worth noting that a class should have only one reason to change. In the context of the delivery client application, for example, any alteration to New York's city rules would only affect its underlying class, leaving the rest of the system untouched.

With the concrete classes for New York and Los Angeles nearly

complete, the final task is to define the **SaoPauloStrategy** class. Its creation adheres to the same principles, and the specific implementation can be viewed in the following *Figure 14.8*:

```
1 reference
public class SaoPauloStrategy : ICityStrategy
{
    2 references
    public void ApplyEmploymentRules()
    {
        Console.WriteLine("Must have permanent employees");
    }

    2 references
    public decimal ApplyProfit(decimal price)
    {
        return price * 0.15m;
    }

    2 references
    public decimal ApplyTax(decimal price)
    {
        return price * 0.10m;
    }
}
```

Figure 14.8: Sao Paulo strategy class

this stage, all the concrete city classes have been constructed, and the requirements specified by the client application have been individually integrated within each corresponding city strategy class. The architecture, as outlined by the strategy design pattern, now necessitates the creation of the context class. This class will be responsible for formulating the correct instance of the city strategy class, thereby relieving the client application of any decision-making duties related to the creation of specific classes for each scenario.

With this in mind, a strategy method class must be devised. Its function is to return the proper concrete class, guided by certain criteria. Within the framework of this chapter, that criterion is the city code information found within the User class. Hence, contingent on the city code, the system is tasked with formulating the precise instance of the city strategy class.

The addition of this auxiliary structure may seem optional but serves to enhance code readability for developers. Furthermore, it

augments the prospects for developing meaningful unit and integration tests specific to the strategy design pattern classes.

Typically, when managing multiple options, a switch-case statement is employed to oversee the creation of the concrete object within the strategy method class. This is particularly relevant when dealing with a multitude of alternatives. In some situations, a more intricate condition might be required to orchestrate the logic behind creating the correct instance. However, in accordance with the requirements of this example, a straightforward switch-case statement suffices, as illustrated in the ensuing [Figure 14.9](#):

```
public class CityStrategyMethod
{
    1 reference
    public ICityStrategy SetCityStrategy(string cityPrefix)
    {
        ICityStrategy cityStrategy;

        switch (cityPrefix)
        {
            case "NYC":
                cityStrategy = new NewYorkStrategy();
                break;
            case "LAC":
                cityStrategy = new LosAngelesStrategy();
                break;
            case "SPC":
                cityStrategy = new SaoPauloStrategy();
                break;
            default:
                cityStrategy = new NewYorkStrategy();
                break;
        }

        return cityStrategy;
    }
}
```

Figure 14.9: Strategy method class

As depicted in [Figure 14.9](#), the class features a singular method that accepts the city code as a parameter. Depending on this code, the switch-case statement navigates to the corresponding city strategy class to formulate the requisite concrete object. If no matching city code is found, the New York city class is instantiated by default, in

accordance with the requirements.

The concluding step for this hands-on exploration involves invoking the classes within the client application. This could take the form of a simple Console application that establishes instances of both user and delivery objects. For educational purposes, it is vital to construct instances that encapsulate the implementation for all the cities defined in this example. Additionally, unit tests can be devised to ensure that the correct profit, tax, and employment rule information is being duly applied by each distinct concrete class.

To initiate the utilization of the classes, you might begin by creating a user object. Specifically, you can define the city of New York, using the code `NYC` as a parameter for the strategy method class, an operation depicted in the subsequent [Figure 14.10](#):

A screenshot of a code editor showing C# code for a client application. The code is enclosed in a dark-themed window with a title bar that says "0 references". The code defines a `Main` method that takes an array of strings as an argument. Inside the method, it creates a `User` object with `CityCode` set to "NYC" and `City` set to "New York". It then creates a `CityStrategyMethod` object and sets its `CityStrategy` property to a new `CityStrategyMethod` object, passing the `user.CityCode` as a parameter. Next, it creates a `Delivery` object, passing the `cityStrategy` object as a parameter, and sets its `Price` to 10.00m. Finally, it calls `cityStrategy.ApplyEmploymentRules()`, prints the tax and profit values to the console, and reads a line from the console.

```
0 references
static void Main(string[] args)
{
    User user = new User();
    user.CityCode = "NYC";
    user.City = "New York";

    CityStrategyMethod cityStrategyMethod = new CityStrategyMethod();
    ICityStrategy cityStrategy = cityStrategyMethod.SetCityStrategy(user.CityCode);

    Delivery delivery = new Delivery(cityStrategy);
    delivery.Price = 10.00m;

    cityStrategy.ApplyEmploymentRules();
    Console.WriteLine($"Tax: {delivery.Tax}");
    Console.WriteLine($"Profit: {delivery.Profit}");

    Console.ReadLine();
}
```

Figure 14.10: *The client application*

The city strategy method object is constructed, and subsequently, a concrete city strategy class is instantiated at runtime. This is achieved by passing the city code to the method `SetCityStrategy`. The following the implementation, a delivery object is created, with the city strategy object supplied as a parameter.

Should you choose to display the values for tax, profit, and employment, the correct values specific to New York City will be retrieved. These figures are consistent with the predetermined requirements for the city regarding employment regulations, tax, and profit, a relationship further illustrated in the subsequent [Figure](#)

14.10.

As highlighted throughout this chapter, employing the strategy pattern infrastructure within the client application promotes simplicity and adheres to best practices in **Object-Oriented Programming (OOP)**. This approach effectively encapsulates the implementation from the target class, upholding a fundamental principle of OOP.

Once the strategy pattern has been fully grasped, it is advisable to extend the practice by creating instances of the user and delivery objects for all conceivable cities illustrated in this example. This expansion allows for the development of unit tests, a critical step to meticulously evaluate and validate the outcomes. Such testing ensures that the application responds appropriately to various scenarios, enhancing its robustness and reliability.

Strategy pattern and Azure Functions

Azure Functions, a serverless compute service provided by Microsoft's Azure platform, aligns closely with the principles of the Strategy Design Pattern. Like the Strategy pattern, Azure Functions encourages the development of modular, independent pieces of functionality that can be switched out or extended without altering the core application's architecture. Each function in Azure is akin to a concrete strategy in the design pattern, encapsulating specific logic that can be executed on-demand, and can be tailored for various scenarios or requirements.

The correlation between Azure Functions and the Strategy Design Pattern emphasizes a flexible, scalable approach to software design. Just as the Strategy Pattern allows for the encapsulation of algorithms and easy swapping between them, Azure Functions enables the creation of isolated, reusable components that can be seamlessly integrated into various parts of an application. This promotes code maintainability, reduces complexity, and offers the agility to adapt to changes. Furthermore, the serverless nature of Azure Functions means that they can scale effortlessly to meet demand, reflecting the adaptable and extensible ethos that is central to the Strategy Design Pattern. It illustrates a modern realization of a classic design principle, tailored to the needs and opportunities of

cloud-native development.

Conclusion

Throughout this chapter, we've explored the strategy pattern, a design approach particularly suited for scenarios where consistent business logic must be applied throughout an application, yet with variations in the rules according to specific business requirements. The wisdom of utilizing this pattern shines through in complex projects, where it helps simplify architectures, enhances maintainability, and empowers developers to extend functionalities safely without impacting existing code.

The practical example provided in this chapter illuminated how to craft a real-world scenario using the strategy pattern, leveraging the C# language. More than just a technical solution, this example showcased a methodology for applying streamlined solutions to intricate problems, embracing the best practices of object-oriented programming.

As you continue to the next chapter, you'll embark on a fresh exploration within the realm of design patterns using the C# language and .NET platform. Get ready to dive into the observer pattern, and build on the foundational knowledge you've cultivated thus far in the book. This next chapter promises to deepen your understanding and further sharpen your skills, ensuring a comprehensive grasp of these essential programming concepts.

Points to remember

- The strategy pattern gives us the possibility of encapsulating from the client application all the logic behind the creation of concrete classes, being a flexible alternative to extend existing functionalities without impacting the legacy code.
- In the cases where the application requires the implementation of a highly divergent set of rules between scenarios, the use of the strategy pattern is not recommended.
- The use of interfaces in the strategy pattern implementation helps us to implement unit and integrations tests and to cover 100% of the cases by testing a single strategy method class. Therefore, the specification of tests is not expensive and has

huge benefits.

Multiple choice questions

1. **In which scenario is the strategy pattern recommended?**
 - A. When objects need to be created in compile time
 - B. When dynamic objects need to be created in runtime
 - C. When the classes share the same requirements
 - D. None of them
2. **Looking at the example given in this chapter, which class does not contain the necessary implementation for the proxy pattern?**
 - A. New York Strategy class
 - B. Los Angeles Strategy class
 - C. City Strategy Method class
 - D. User class

Answers

	B	
	D	

Questions

1. Explain the primary purpose of the strategy pattern.
2. According to what you have learned in this chapter, create a hypothetical scenario in which the strategy pattern can be applied, apart from the example given during this chapter.
3. Explain the type of scenarios where the strategy pattern can be used.
4. Are there any projects that you are currently working on that implement the strategy pattern?

Key terms

- **Compile time:** Any operations that can be identified and recognized before the project runs.
- **Runtime:** A specific behavior of the application will only be identified when the application is actually running.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 15

Observer Pattern

Introduction

Modern applications often necessitate the use of event-driven notifications to align with specific business requirements. This demands a robust software architecture that ensures timely and accurate notifications to all subscribers of a particular event. The observer design pattern, detailed in this chapter, offers a valuable solution to this challenge. It facilitates the creation of routines that not only detect changes in an object but also enable us to notify all relevant users or client applications of these changes promptly and accurately.

Grasping the observer pattern opens up opportunities to recognize its applications within legacy projects developed with the C# language and .NET platform. Moreover, it equips you with the ability to incorporate this pattern into a wide array of applications, spanning various platforms such as mobile, desktop, and web.

This chapter serves as your hands-on guide to applying the observer pattern. Through a practical example utilizing the C# language, you will construct a real-world scenario step by step. Whether you're refining existing systems or building new ones, the insights and skills gleaned from this chapter will contribute significantly to your

ability to create more responsive and interconnected applications.

Structure

In this chapter, we will discuss the following topics:

- Observer pattern concept
- Examples in C# and .NET
- Practical usage of Azure Functions for Strategy pattern

Objectives

Upon completing this unit, you will have gained a comprehensive understanding of the observer pattern. This foundational knowledge will enable you to grasp the underlying mechanics of the pattern, its applications, and its potential to create a responsive system that responds to changes within objects.

Furthermore, you will be equipped to identify the observer pattern's presence and implementation within .NET applications and existing projects. This ability to recognize the pattern will prove invaluable as you work on or analyze various projects, allowing you to see how the observer pattern can be employed to enhance the overall structure and functionality.

Lastly, this unit prepares you to apply the observer pattern in real-world scenarios. Whether you're working on mobile, web, or desktop applications, the practical skills acquired through this unit will help you utilize the observer pattern effectively, tailoring it to fit specific requirements and thereby creating more dynamic and interconnected systems.

Observer pattern definition

Nowadays, the ubiquity of applications across multiple devices and platforms has necessitated real-time notifications, enhancing user experiences. Particularly in mobile applications, notifications have become a standard feature, implemented in myriad ways. This represents a common challenge faced globally by developers; however, the solution doesn't necessitate reinvention but rather adaptation to unique scenarios.

Since the advent of modern programming languages, an event-driven architecture has been widely utilized. This approach typically involves applications using events to trigger processes between distinct components or even different services. The reaction of an application to a specific event is not a novel concept in software development; it has been intrinsically employed for many years.

Social media applications, with their vast user bases, have further complicated the challenge of notifications. A single social media platform may have hundreds of events that require notification delivery, each subject to complex rules, permissions, and conditions. In this context, employing a straightforward and common pattern becomes essential. After all, the core of such applications is not solely about notifications, and a robust software architecture must support this crucial feature without compromising other functionalities.

The observer design pattern provides an elegant and efficient solution for implementing applications that require notification features. It is widely used in the C# language and .NET applications, with its logic revolving around having subscribers associated with a specific object. If an object needs to receive notifications based on changes in another object, the notifying object must facilitate the subscription process.

To gain a clear understanding of how the observer pattern functions, one can refer to [Figure 15.1](#). This diagram outlines the basic structure of the pattern and illustrates the relationship between the class that needs to receive notifications (**Observer**) and the class responsible for initiating those notifications (**Subject**). This visualization aids in comprehending the interplay between components, shedding light on why the observer pattern has become a favored solution for many developers working on interconnected and responsive systems:

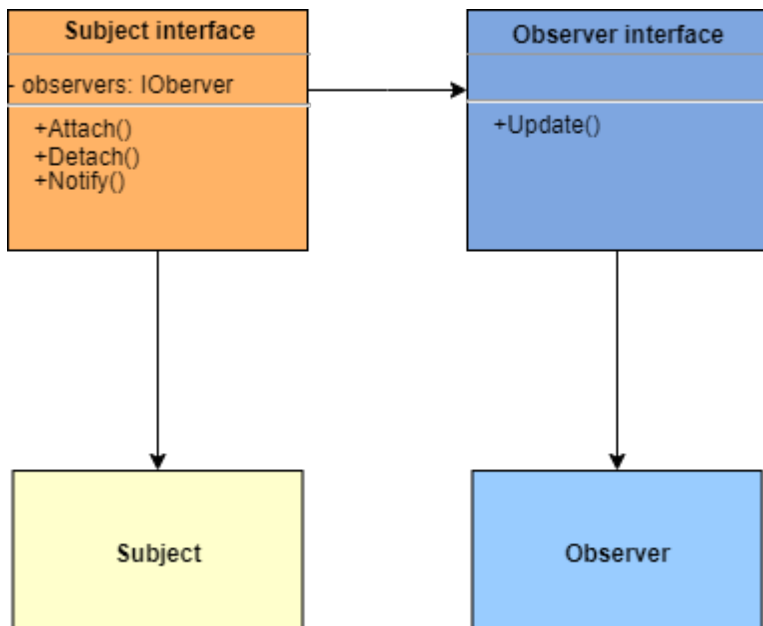


Figure 15.1: Observer pattern diagram

The **Subject** class serves as a central hub, containing information that other classes may wish to track, particularly if changes occur within its state or contents. Accordingly, the concrete **Subject** class houses a private property responsible for maintaining a list of subscribers. These subscribers are essentially objects that express interest in receiving notifications. To manage this list, the class includes methods for attaching and detaching observers, commonly referred to as `subscribe` and `unsubscribe` methods in real-world applications.

As depicted in [Figure 15.1](#), the **Observer** interface features a method named `Update`. This method must be implemented by all concrete observer classes and serves a crucial role. Specifically, the `Update` method is called by the **Subject** class within its `Notify` method for each individual object listed in the observer collection. Through the judicious use of interfaces, the observer design pattern's implementation permits various concrete classes to subscribe to notifications. This grants the pattern both flexibility and extensibility, attributes that can prove highly beneficial in complex systems.

Later in this chapter, you will have the chance to put theory into practice by constructing a real-world application utilizing the observer pattern from the ground up. The example provided takes the form of a blog application, where users can opt to subscribe to notifications. In this context, notifications would alert subscribers to the creation of a new post within the blog. This hands-on exercise not only reinforces the concepts explored but also demonstrates how the observer pattern can be applied to common development scenarios.

Observer pattern implementation

As with the previous chapters covering different design patterns, this chapter will guide you through the practical implementation of the observer design pattern, which was explained theoretically earlier on. This hands-on experience will enhance your ability to recognize the pattern's usage in existing projects and empower you to apply it in various contexts. Particularly, you'll find the pattern useful in scenarios requiring the implementation of notifications or any event-driven routines.

For the practical example, we'll be creating a blog application that allows users to subscribe to different blogs. Upon subscription, they will receive notifications whenever a new post is created within the blogs they follow. This dynamic application can host multiple micro-blogs, managed by various users, and facilitate multiple subscriptions by different users to a wide array of blogs.

To begin, we must define the interfaces common to any application utilizing the observer design pattern. While the context may differ across diverse systems, the foundational concepts of subject and observer remain consistent. Therefore, we'll start by creating the `ISubject` interface, an integral part of the pattern, as illustrated in the subsequent [Figure 15.2](#). This interface lays the groundwork for the pattern's application within our specific example, while also showcasing its flexibility for different applications:

```

1 reference
public interface ISubject
{
    4 references
    void Attach(IObserver observer);
    1 reference
    void Detach(IObserver observer);
    2 references
    void Notify();
}

```

Figure 15.2: Subject interface

The **ISubject** interface plays a crucial role in defining the relationship between subjects and observers. It includes two essential methods specifically designed for managing observers in the subject's concrete classes. These methods handle the addition and removal of observers, enabling dynamic control over who receives notifications. Additionally, the interface mandates a **Notify** method, compelling the concrete class to formulate the unique logic governing the notification process, as this may vary across different classes.

Complementing the **ISubject** interface, the **IObserver** interface must also be defined. It sets the contractual obligations for all **observer** classes, ensuring a standardized approach to receiving notifications. This uniformity enhances the pattern's flexibility and extensibility, allowing it to be adapted to various scenarios with ease. A representation of these key interfaces and their relationships is provided in the subsequent [Figure 15.3](#):

```

- namespace BPBDesignPatternsChapter15
{
    7 references
    public interface IObserver
    {
        2 references
        void Update(object obj);
    }
}

```

Figure 15.3: Observer interface

The **ISubject** interface plays a crucial role in defining the relationship between subjects and observers. It includes two essential methods specifically designed for managing observers in the subject's concrete classes. These methods handle the addition and removal of observers, enabling dynamic control over who receives notifications. Additionally, the interface mandates a **Notify** method, compelling the concrete class to formulate the unique logic governing the notification process, as this may vary across different classes.

Complementing the **ISubject** interface, the **IObserver** interface must also be defined. It sets the contractual obligations for all observer classes, ensuring a standardized approach to receiving notifications. This uniformity enhances the pattern's flexibility and extensibility, allowing it to be adapted to various scenarios with ease. A representation of these key interfaces and their relationships is provided in the subsequent [Figure 15.4](#):

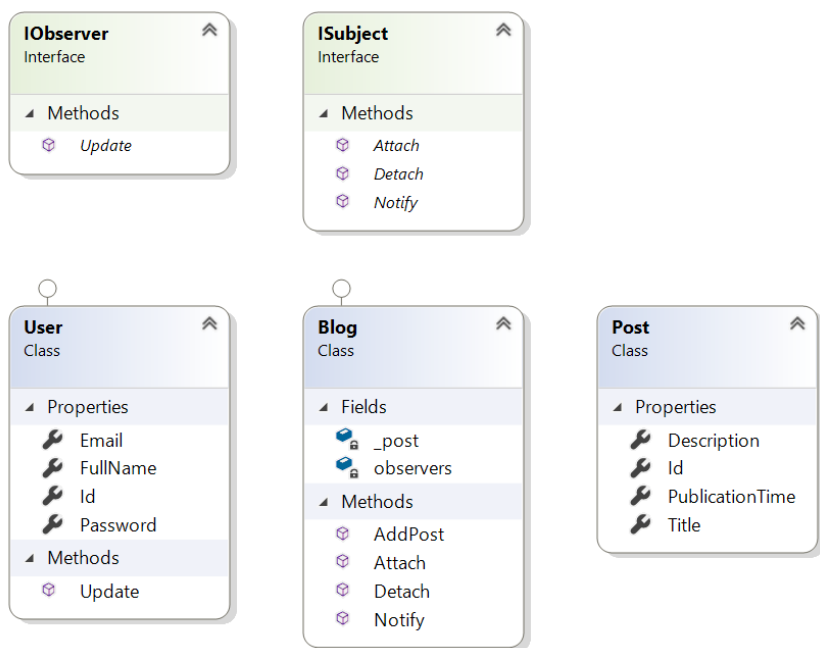


Figure 15.4: Class diagram

In this particular scenario, the **Blog** class takes on the role of the subject and therefore implements the **ISubject** interface. Conversely, the **User** class adopts the observer role, fulfilling the requirements of the observer interface. This clear division of responsibilities facilitates a modular design, allowing each class to focus on its specific function within the overall pattern.

Creating the User class

The implementation of the concrete User class is illustrated in the accompanying *Figure 15.5* below:



```
6 references
public class User : IObserver
{
    0 references
    public int Id { get; set; }
    4 references
    public string FullName { get; set; }
    0 references
    public string Email { get; set; }
    0 references
    public string Password { get; set; }

    2 references
    public void Update(object obj)
    {
        var post = (Post)obj;

        Console.WriteLine($"User: {this.FullName} - New Post: {post.Title} ");
    }
}
```

Figure 15.5: User class

In the implementation of the **User** class, the **Update** method receives a generic object as a parameter and casts it to a specific **Post** object. This method is invoked by the subject class whenever a notification event occurs. Utilizing a generic object for the notification provides significant flexibility, as various classes may anticipate receiving different types of notifications.

As previously illustrated, the **Update** method displays the username and the title of the blog post on the screen for demonstration purposes. This immediate feedback allows you to track the activity within the observer pattern implementation. In a real-world context, this same method could be employed to present an inbox notification to users within a mobile application or to refresh the display with the details of the new blog post.

Conversely, the **Blog** class functions as the Subject class within this context and must adhere to the related interface. This requires the tailored specification for methods handling the attachment and detachment of observers, as well as the implementation of the method responsible for notifying all subscribed observers. For the sake of clarity in this demonstration, the **Blog** class is structured as depicted in the accompanying [Figure 15.6](#):

```
2 references
public class Blog : ISubject
{
    private Post _post = new Post();
    private List<IObserver> observers = new List<IObserver>();
    4 references
    public void Attach(IObserver observer)
    {
        observers.Add(observer);
    }

    1 reference
    public void Detach(IObserver observer)
    {
        observers.Remove(observer);
    }

    2 references
    public void Notify()
    {
        foreach (var observer in observers)
        {
            observer.Update(_post);
        }
    }

    1 reference
    public void AddPost(Post post)
    {
        _post = post;
        Notify();
    }
}
```

Figure 15.6: Blog class

As depicted in the preceding [Figure 15.6](#), the **Attach** and **Detach** methods solely undertake the task of modifying the private observer list within the class. These methods are vital for managing subscription events that may transpire within the client application. Adhering to the principles of the observer design pattern, the **Notify** method systematically traverses the observer list, executing the **Update** method on each observer object.

In conclusion, the **Notify** method is invoked by the **AddPost** method, which is triggered every time a new post is added to the blog. This close integration between the **AddPost** and **Notify** methods embodies the event-driven development philosophy, endowing the application with dependable behavior concerning its core function of dispatching notifications when a new post is generated.

Creating the Blog Post class

The implementation of the blog post can be viewed in [Figure 15.7](#) provided in the following:

```
6 references
public class Post
{
    0 references
    public int Id { get; set; }
    2 references
    public string Title { get; set; }
    1 reference
    public string Description { get; set; }
    1 reference
    public DateTime PublicationTime { get; set; }
}
```

Figure 15.7: Post class

In cloud applications, particularly when managing notifications across a vast number of devices during critical demand processes, the concept of a queue is widely used. Real-time notifications are a staple of modern development, and the .NET platform has embraced this with SignalR technology. SignalR enables real-time communication between client and server, adhering to the subscriber concept outlined by the observer design pattern.

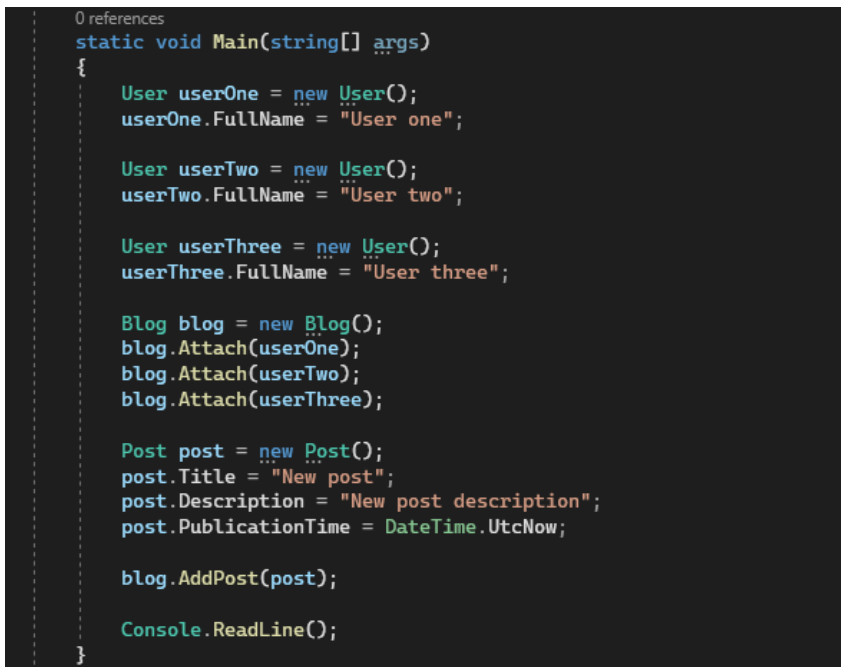
Within this context, the client application, whether a web page opened in a browser or a desktop application, takes on the role of the subject in the observer pattern. Should there be any changes in the client application, the server must be notified. Subsequently, all necessary alterations must be synchronized across the myriad devices subscribed to a specific event. SignalR represents a powerful technology that greatly simplifies the development of real-time

applications, such as chat platforms or real-time interactive screens.

At this juncture, you are equipped with all the necessary infrastructure to implement and thoroughly test the observer pattern, utilizing hypothetical data. Given the custom implementation of the **Update** method within the **User** class, you can scrutinize the output to fully understand the precise behavior and effectiveness of your solution.

Client application

To accurately emulate a real-world scenario, you can proceed by manually creating multiple instances of the **User** class. Next, you'll need to create a new instance of the **Blog** class. Once these instances are created, the final step involves attaching the user objects to the **Blog** class. The process for doing so is depicted in the following [Figure 15.8](#):



```
0 references
static void Main(string[] args)
{
    User userOne = new User();
    userOne.FullName = "User one";

    User userTwo = new User();
    userTwo.FullName = "User two";

    User userThree = new User();
    userThree.FullName = "User three";

    Blog blog = new Blog();
    blog.Attach(userOne);
    blog.Attach(userTwo);
    blog.Attach(userThree);

    Post post = new Post();
    post.Title = "New post";
    post.Description = "New post description";
    post.PublicationTime = DateTime.UtcNow;

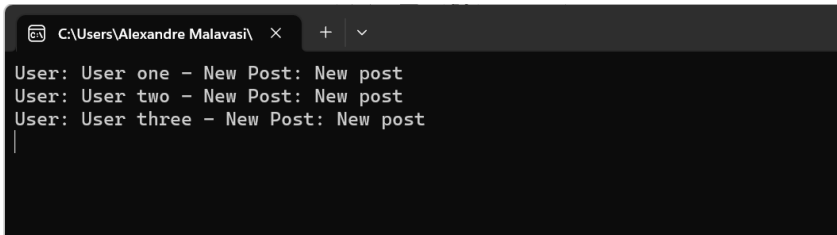
    blog.AddPost(post);

    Console.ReadLine();
}
```

Figure 15.8: The client application

The preceding screenshot, showcasing a Console application, provides a step-by-step demonstration of the process. Initially, three

user objects are created, named **userOne**, **userTwo**, and **userThree**. Following this, each of these objects is attached to the blog object. Subsequently, a new post is crafted and added to the blog. If you execute the application, the resulting output will be illustrated in the following [Figure 15.9](#):



```
C:\Users\Alexandre Malavasi\ >
User: User one - New Post: New post
User: User two - New Post: New post
User: User three - New Post: New post
```

Figure 15.9: Client application result

To grasp the outcome of the implementation accurately, it's crucial to recall the sequence of actions the application performs, as detailed below:

1. Create new instances of the **User** class.
2. Instantiate a new object of the **Blog** class.
3. Attach the user objects to the **Blog** class, thereby subscribing them to receive notifications whenever a new post is created.
4. Add a new post object to the blog using the relevant method within the **Blog** class.
5. The **AddPost** method triggers the **Notify** method, subsequently calling the **Update** method within the user class. The specific implementation of this part can be seen in the following [Figure 15.10](#):



```
6 references
public class User : IObserver
{
    0 references
    public int Id { get; set; }
    4 references
    public string FullName { get; set; }
    0 references
    public string Email { get; set; }
    0 references
    public string Password { get; set; }

    2 references
    public void Update(object obj)
    {
        var post = (Post)obj;
        Console.WriteLine($"User: {this.FullName} - New Post: {post.Title} ");
    }
}
```

Figure 15.10: Update method

As demonstrated throughout this chapter, implementing the observer design pattern requires alterations both in the class tasked with sending the notification and in the class receiving it. To facilitate identification and maintenance of this pattern, it is strongly advised to employ descriptive and intuitive names for all interfaces and components involved. Doing so not only aids other developers during the project's maintenance phase but also ensures that the underlying principles of this design pattern are adhered to, particularly if future extensions to the application are needed.

We hope that this chapter has equipped you with a solid understanding of the observer pattern. Building upon the insights you've gained through this practical illustration, I encourage you to explore and create additional scenarios that utilize the observer pattern. Experimenting beyond the example provided here will offer invaluable hands-on experience and deepen your mastery of this essential design concept.

Conclusion

In this chapter, you've delved into the observer pattern, a cornerstone in software development, particularly when an application necessitates the transmission of notifications to multiple objects.

The observer pattern is not just a theoretical concept; it's a pragmatic design pattern widely utilized in today's applications. Its relevance extends to varied real-world scenarios, such as managing notifications in social media applications, across different devices and vast user bases. This pattern stands out as one of the most popular tools in a developer's toolkit.

Through hands-on examples using the C# language, this chapter has guided you through the intricacies of implementing the observer pattern, while adhering to best practices including the SOLID principles and embracing the object-oriented programming paradigm.

As you progress to the next chapter, you'll be introduced to a concise overview of other prevalent design patterns employed in .NET applications, ones that haven't been explored in previous

chapters of this book. This upcoming section will also enlighten you with vital practices and recommendations for effectively applying these design patterns in real-world applications. By connecting theoretical concepts with practical implementations, this book aims to equip you with the skills and knowledge needed to excel in modern software development.

Points to remember

- The observer pattern allows implementing the necessary logic to send a notification to multiple objects based on events.
- It is highly recommended to use interfaces in the observer design pattern implementation in order to have a more flexible and extensible software architecture.

Multiple choice questions

1. **According to what you have learned and the practical example given in this chapter, in which class should the Notify method be implemented?**
 - A. In the subject class
 - B. In a static class
 - C. In an external service
 - D. In the observer class
2. **Which class represents the observer class in the practical example given in this chapter?**
 - A. The blog class
 - B. The post class
 - C. The user class
 - D. None of them

Answers

Questions

1. Explain the primary purpose of the observer pattern.
2. According to what you have learned in this chapter, create a hypothetical scenario in which the observer pattern could be applied apart from the example given during this chapter.
3. Explain in which type of scenarios the observer pattern can be used.
4. Are there any projects that you are currently working on that implement the observer pattern?

Key terms

- **Observer class:** The class that expects to receive notifications.
- **Subject class:** The class responsible for sending the notification to all subscribed objects.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

- Abstract Factory Pattern [145](#)
 - definition [146](#), [147](#)
 - implementation [149-160](#)
 - scenario [147-149](#)
- access modifiers [78-80](#)
- adapter pattern [197](#)
 - definition [199](#)
 - diagram [200](#)
 - implementation [200-206](#)
- arithmetic operators [29](#)
- ASP.NET Core Web App [165](#), [166](#)
- ASP.NET Model View Controller (MVC) project [13](#)
- ASP.NET Web Forms [13](#)
- assignment operator [30](#)
- Azure Function [57-70](#)
- Azure Functions [280](#), [281](#)
 - correlation, with Strategy pattern [280](#)
- Azure storage accounts [52-57](#)

B

- behavioral design patterns [121](#)
- blog post class, observer pattern
 - creating [291](#), [292](#)

C

- C#
 - tools and environment setup [2](#)
- C# 10
 - features [41](#)
 - Generic Attribute class [42](#), [43](#)
 - global, using directives [41](#), [42](#)
 - list patterns [43](#)
- C# 11
 - features [41](#)
 - raw string literals [43](#), [44](#)
- C# 11 features [191](#)

- list patterns [193](#), [194](#)
- newlines, in string interpolations [193](#)
- raw string literals [194](#)
- required fields [191](#), [192](#)
- UTF-8 string literals [193](#)
- city strategy class
 - creating [273-280](#)
- C# language [18](#), [19](#)
 - do-while statement [26](#), [27](#)
 - error handling [38-40](#)
 - foreach statement [28](#)
 - for loop [27](#)
 - if statement [35](#), [36](#)
 - loops [25](#)
 - namespace [40](#), [41](#)
 - nested if clauses [37](#)
 - switch case statement [36](#), [37](#)
 - variables [20](#)
 - while statement [26](#)
- client application, observer pattern [292-294](#)
- Command Pattern [247](#)
 - Client application implementation [261](#)
 - client application, with transaction [263](#)
 - common interface class [255](#)
 - CreditCard class [254](#)
 - customer and credit card objects [260](#)
 - definition [248-250](#)
 - implementation [250-253](#)
 - MediatR integration [264](#)
 - Payment command class [256](#)
 - Product class [253](#)
 - PurchaseLogic class [259](#)
 - storage command class [257](#)
 - Supplier class [253](#)
 - supplier command implementation [258](#)
 - transaction purchase command class [262](#)
- Common Intermediate Language (CIL) [13](#)
- Common Language Runtime (CLR) [12](#)
- composite pattern [209](#)
 - definition [210](#), [211](#)
 - hierarchical representation [212](#)
 - implementation [213-221](#)
- composites [212](#)
- compound assignment operators [34](#), [35](#)
- constructor, OOP [87](#), [88](#)
- ContentRepository class [233](#)
- creational patterns [120](#), [121](#)

D

debugger features, in Visual Studio
 breakpoint hover glyph [45](#)
 temporary breakpoints [44](#), [45](#)
dependency injection [136-141](#)
dependency inversion principle (DIP) [112-114](#)
design patterns [118](#), [119](#)
 behavioral patterns [121](#)
 creational patterns [120](#), [121](#)
 for distributed systems [122](#), [123](#)
 history [119](#), [120](#)
 microservices design patterns [123](#), [124](#)
 structural patterns [122](#)
do-while statement [26](#), [27](#)

E

equal operator [30](#)
error handling, C# [38-40](#)
exception handling [39](#)
Extensible Application Markup Language (XAML) [14](#)
Extensible Markup Language (XML) [14](#)

F

Factory Method pattern
 definition [180](#), [181](#)
 implementation [181-190](#)
finally block [40](#)
foreach statement [28](#), [29](#)
for loop [27](#), [28](#)

G

Gang of Four (GoF) [119](#)
Generic type class [23](#)
 example [24](#)
GRPC [239](#)
 with Blazor [239-244](#)

H

hot reload [71](#)
HTTP/3 [72](#)

I

IContentRepository interface [234](#)
if statement [35](#), [36](#)

Integrated Development Environment (IDE) [2](#)
interface, OOP [92-95](#)
interface segregation principle [110-112](#)
Intermediate Language (IL) [13](#)
Internet of Things (IoT) [50](#)

J

JavaScript Object Notation (JSON) [14](#)
JSON Web Token (JWT) [230](#)

L

leaf [212](#)
Liskov substitution principle [107-110](#)
logical operators [32](#), [33](#)

M

MediatR [264](#)
 integrating, with Command pattern [264](#)
microservices design patterns [123](#), [124](#)
MobilePlan [184](#)
Movie class [83](#)
multi-platform concepts [50](#), [51](#)

N

namespace, C# [40](#), [41](#)
.NET
 fundamentals [49](#)
 history [12-15](#)
 .NET 5 [18](#)
 .NET 6 [18](#)
 .NET 7 [18](#)
 .NET Core versions [15-17](#)
 tools and environment setup [2](#)
.NET 6
 custom platform checks [73](#)
 features [71](#)
 hot reload [71](#)
 HTTP/3 [72](#), [73](#)
 .NET MAUI [71](#), [72](#)
.NET, for cloud development [51](#)
 Azure Function [57-70](#)
 Azure storage accounts [52-57](#)
.NET MAUI [71](#), [72](#)

O

Object-Oriented Programming (OOP) [77](#), [78](#)

- access modifiers [78-80](#)

- classes [78-80](#)

- constructor [87](#), [88](#)

- encapsulation [81](#)

- inheritance [81](#), [82](#)

- interfaces [92-95](#)

- polymorphism [83](#), [84](#)

- reusability [82](#)

- static classes [88-90](#)

- structs [90](#), [91](#)

Object-Relational Mapping (ORM) [15](#)

- observer pattern [283](#)

- blog post class, creating [291](#), [292](#)

- client application [292-294](#)

- definition [284-286](#)

- implementation [286-288](#)

- user class, creating [289-291](#)

open-closed principle (OCP) [100-106](#)

Open Database Connectivity (ODBC) [13](#)

- operators [29](#)

- arithmetic operators [29](#)

- assignment operator [30](#)

- compound assignment operators [34](#), [35](#)

- equal operator [30](#)

- logical operators [32](#), [33](#)

- relational operators [31](#), [32](#)

- ternary operators [34](#)

- unary operators [33](#), [34](#)

P

- partial class [85](#), [86](#)

- Product class [84](#)

- Product parent class [83](#)

- Prototype Pattern [163](#)

- definition [164](#), [165](#)

- implementation [165-176](#)

- Proxy pattern

- cache [227](#)

- Client application [228](#)

- definition [226](#)

- diagram [228](#)

- implementation [230](#)

- practical example [231-238](#)

- protection [227](#)

- Proxy layer [228](#)

- Real Service [228](#)

- remote [227](#)
- scenario requirements [231](#)
- smart [226](#)
- structure [227-229](#)
- types [226](#), [227](#)
- using, in real projects [229](#), [230](#)
- virtual [227](#)

R

- reference type variables, C#
 - array [22](#)
 - list [23](#)
 - other types [25](#)
- relational operators [31](#), [32](#)

S

- Shallow Copying [165](#)
- Single Page Application (SPA) development [239](#)
- single pattern
 - definition [128-130](#)
 - modifications [131-135](#)
 - multiple instances [130](#), [131](#)
 - thread-safe implementation [135](#), [136](#)
- Single Responsibility Principle (SRP) [87](#), [98-100](#)
- Software As a Service (SaaS) model [50](#)
- SOLID principles
 - dependency inversion principle [112-114](#)
 - interface segregation principle [110-112](#)
 - Liskov substitution principle [107-110](#)
 - open-closed principle [100-106](#)
 - Single Responsibility Principle [98-100](#)
- static class, OOP [88-90](#)
- Strategy pattern [267](#)
 - city strategy class, creating [273-280](#)
 - correlation, with Azure Functions [280](#), [281](#)
 - definition [268](#), [269](#)
 - implementation [271-273](#)
 - structure [269](#), [270](#)
- structs, OOP [90](#), [91](#)
- structural patterns [122](#)
- switch case statement [36](#), [37](#)

T

- ternary operators [34](#)

U

unary operators [33](#), [34](#)

User class, observer pattern
creating [289-291](#)

V

variables, C# language

boolean [20](#)

byte [21](#)

char [21](#)

decimal [21](#)

examples [21](#), [22](#)

float [21](#)

int [20](#)

object [20](#)

reference types [22](#)

string [21](#)

value types [22](#)

Visual Studio

debugger features [44](#)

Visual Studio 2022 [5](#)

installing [2](#), [3](#), [4](#)

project, creating [5-10](#)

templates [6](#)

Visual Studio Code [4](#), [10-12](#)

installing [4](#), [5](#)

W

Web Application Programming Interface (API) model [14](#)

while statement [26](#)

Windows Communication Foundation (WCF) [14](#)

Windows Presentation Foundation (WPF) [14](#)